Typage polymorphe d'un langage algorithmique

Xavier Leroy

THÈSE DE DOCTORAT

présentée à

L'UNIVERSITÉ PARIS 7

Spécialité : Informatique

par

Xavier LEROY

Sujet de la thèse :

Typage polymorphe d'un langage algorithmique

Soutenue le 12 juin 1992 devant la Commission d'examen composée de

MM.	**Jean-Pierre BANÂTRE**	**Président**
	Guy COUSINEAU	**Rapporteurs**
	Matthias FELLEISEN	
	Pierre CASTERAN	**Examinateurs**
	Gérard HUET	
	Gilles KAHN	
	Michel SINTZOFF	

Résumé

Le typage statique avec types polymorphes, comme dans le langage ML, s'adapte parfaitement aux langages purement applicatifs, leur apportant souplesse et expressivité. Mais il ne s'étend pas naturellement au trait principal des langages algorithmiques : la modification en place des structures de données. Des difficultés de typage similaires apparaissent avec d'autres extensions des langages applicatifs : les variables logiques, la communication inter-processus à travers des canaux, et la manipulation de continuations en tant que valeurs.

Ce travail étudie, dans le cadre de la sémantique relationnelle, deux nouvelles approches du typage polymorphe de ces constructions non-applicatives. La première repose sur une restriction de l'opération de généralisation des types (la notion de variables dangereuses), et sur un typage plus fin des valeurs fonctionnelles (le typage des fermetures). Le système de types obtenu reste compatible avec le noyau applicatif de ML, et se révèle être le plus expressif parmi les systèmes de types pour ML plus traits impératifs proposés jusqu'ici. La seconde approche repose sur l'adoption d'une sémantique "par nom" pour les constructions du polymorphisme, au lieu de la sémantique "par valeur" usuelle. Le langage obtenu s'écarte de ML, mais se type très simplement avec du polymorphisme. Les deux approches rendent possible l'interaction sans heurts entre les traits non-applicatifs et le typage polymorphe.

Mots-clés

Typage. Polymorphisme. ML. Langages applicatifs. Références. Canaux de communication. Continuations. Variables dangereuses. Typage des fermetures. Polymorphisme par nom. Sémantique naturelle. Sémantique opérationnelle structurée. Sûreté du typage.

Remerciements

M. Jean-Pierre Banâtre me fait l'honneur de présider le jury de cette thèse ; je l'en remercie.

Ma gratitude va à MM. Guy Cousineau et Matthias Felleisen, qui ont accepté la lourde charge d'être les rapporteurs de cette thèse un peu longue et fastidieuse parfois. Je les remercie pour leur lecture attentive et leurs commentaires pertinents.

Malgré l'obstacle de la langue, M. Matthias Felleisen a décortiqué avec soin ce travail, et me l'a fait voir sous des angles nouveaux. J'ai beaucoup apprécié les discussions animées et transatlantiques que j'ai eues avec lui à cette occasion.

La contribution de M. Guy Cousineau à ce travail va bien au-delà de son rôle de rapporteur. Son enseignement en première année de l'École Normale Supérieure m'a fait changer d'avis sur l'Informatique, à une époque où j'en connaissais ce que j'en avais vu dans *Byte Magazine*, et me destinais par conséquent aux Mathématiques. Quelques mois plus tard, c'est encore lui qui, alors que je cherchais un stage "de programmation système ou peut-être de compilation", m'a fait découvrir le langage CAML, me signalant qu'il y avait là "un gros travail de compilation à faire", et déterminant ainsi l'orientation de mon travail pendant les quatre années qui ont suivi.

M. Gérard Huet a dirigé mon travail de thèse. Tout en me laissant entière liberté sur le plan scientifique, il m'a donné à point nommé des conseils précieux sur le déroulement de ce travail, me rappelant opportunément que le but de cette thèse n'était pas de rendre compte de tout ce que j'avais fait pendant ces trois années de thèse, et me signalant que le sujet des références polymorphes pourrait faire une thèse complète à l'époque où je l'imaginais tout juste digne d'un chapitre dans une œuvre beaucoup plus vaste.

Je suis très heureux que M. Gilles Kahn fasse partie du jury de cette thèse. Ce travail s'inscrit résolument dans le cadre de la sémantique naturelle que lui et son équipe ont mis sur pied. Dans quelques années, se dégagera sans doute une génération d'étudiants en Informatique qui ont commencé à comprendre la pratique de la sémantique en lisant "Mini-ML".

M. Pierre Casteran et M. Michel Sintzoff ont bien voulu faire partie de ce jury. Je les remercie de l'intérêt qu'ils portent à mon travail.

Cette thèse s'inscrit dans le prolongement des travaux de M. Mads Tofte. C'est en lisant sa thèse que j'ai compris comment mener les preuves de sûreté du typage ; j'ai repris et adapté plusieurs de ses techniques. Je le remercie tout particulièrement pour la clarté et la pédagogie de ses écrits.

Cette thèse doit beaucoup à la présence de M. Didier Rémy. Il a suggéré la représentation indirecte des types qui est au cœur du chapitre 4, et fourni son expertise en matière d'inférence de types. Pendant les moments difficiles, il m'a bien des fois patiemment écouté lui exposer (un peu hystériquement) pourquoi "ça ne pouvait pas marcher" ; à chaque fois, il a su (avec le plus grand calme) m'indiquer des possibilités nouvelles. Je lui dois enfin une relecture remarquablement attentive de ce document.

Les idées de base de ce travail se sont dégagées au cours de discussions avec M. Pierre Weis. C'est en collaboration avec lui que la première version publiée de ce travail a été rédigée ; elle y a gagné en clarté et en densité de l'exposition.

C'est en étudiant les travaux de M. Luca Cardelli que j'ai compris l'importance du polymorphisme par nom, en premier lieu, et plus généralement bien des points de "philosophie" du typage statique. Je le remercie également de m'avoir mis le pied à l'étrier en matière de publications.

M. Jean-Pierre Talpin a fait régner sur ce travail une ambiance de saine émulation, me motivant pour mener à bout certaines parties techniques ; je lui en sait gré.

Je tiens à souligner l'extraordinaire environnement dont j'ai bénéficié à l'INRIA Rocquencourt, dans le projet Formel et dans les projets frères, tout au long de ce travail. Un grand merci à tous leurs membres pour les excellents moments passés en leur compagnie.

La réalisation pratique de cette thèse a fait largement appel aux outils généreusement fournis à la collectivité par MM. Richard Stallman, Donald Knuth, Leslie Lamport et Larry Wall.

Introduction

Quand j'étais enfant, on m'avait dit que le Père Noël descendait par la cheminée, et que les ordinateurs se programmaient en binaire. J'ai appris depuis que la programmation se faisait de préférence dans des langages de haut niveau, plus abstraits et plus expressifs. J'ai aussi appris que les programmeurs n'étaient pas infaillibles, et que donc les langages de programmation évolués, non contents de véhiculer la pensée du programmeur, devaient aussi permettre la détection automatique de certaines incohérences dans les programmes.

La plus répandue de ces vérifications de cohérence s'appelle le typage statique. Elle consiste à détecter une vaste famille d'erreurs, celles où une opération du programme est appliquée à des objets sur lesquels elle n'est pas définie (l'addition entière, à un booléen et à une chaîne de caractères, par exemple). Ce but est atteint en regroupant les objets manipulés par le programme en classes : les types, et en simulant abstraitement l'exécution du programme au niveau des types, suivant pour ce faire un ensemble de règles qu'on appelle système de types.

Le point fort du typage statique est qu'il garantit l'absence d'erreurs de cette famille dans les programmes acceptés. Le point faible du typage statique est qu'il rejette des programmes en fait non erronés, mais trop complexes pour avoir été reconnus comme tels par le système de types utilisé. De cette tension s'ensuit la recherche incessante de systèmes de types toujours plus fins [14, 66], vaste quête dans laquelle s'inscrit le travail présenté ici.

Dans cette quête, une étape importante a été franchie avec l'apparition du concept de typage polymorphe, permettant le typage statique de fonctions génériques : des morceaux de programmes qui peuvent opérer sur des données de différents types [82, 58]. Par exemple, un algorithme de tri s'applique à n'importe quel type de tableau dès lors qu'une fonction de comparaison entre éléments est fournie. Ces fonctions génériques sont d'un grand intérêt, car elles sont naturellement réutilisables sans modifications d'un programme à un autre. Sans polymorphisme, le typage statique empêche cette réutilisation ; avec l'introduction du polymorphisme, le typage statique permet cette réutilisation, tout en garantissant qu'elle est bien légitime du point de vue des types.

Les systèmes de types polymorphes s'adaptent parfaitement bien aux langages de programmation purement applicatifs : les langages où il n'y a pas d'affectation sur les variables, et où les structures de données ne peuvent pas être modifiées en place [47, 93, 38]. Au contraire, dans cette Thèse, je m'intéresse au typage polymorphe de langages non purement applicatifs, comme, en premier lieu, les langages algorithmiques de la famille d'Algol : Pascal, Ada, Modula-3, et tout particulièrement ML. Ces langages permettent l'affectation des variables et la modification en place de

structures de données. C'est ce trait qui ne se type pas facilement avec des types polymorphes. Les structures modifiables admettent naturellement des règles de typage très simples ; mais ces règles, bien que correctes dans les systèmes de types monomorphes, se révèlent incorrectes en présence de types polymorphes. Voici un exemple, écrit en ML[1], qui illustre ce fait :

```
let r = ref [] in
    r := [1];
    if head(!r) then ... else ...
```

L'application naïve des règles de typage conduit à donner le type polymorphe $\forall\alpha.\,\alpha$ `list ref` à la variable `r` dans le corps de la construction `let`. Ce type permet d'utiliser `r` une première fois avec le type `int list ref`, et donc d'y stocker la liste contenant l'entier 1. Ce type permet aussi d'utiliser `r` une deuxième fois avec le type `bool list ref`, et donc de traiter `head`(`!r`) (le premier élément de la liste contenue dans `r`) comme un booléen. La construction `if` est donc bien typée. Pourtant, à l'exécution, `head`(`!r`) s'évalue en la valeur 1, qui n'est pas une valeur booléenne. On voit sur cet exemple que la modification en place d'un objet polymorphe peut rendre invalides des hypothèses de typage, et donc compromettre la sûreté du typage : après remplacement de [] par [1] dans `r`, l'hypothèse `r` : $\forall\alpha.\,\alpha$ `list ref` n'est plus vérifiée.

Comme on vient de le montrer, un système de types polymorphes doit, pour être correct, restreindre l'emploi des objets modifiables polymorphes. Un même objet modifiable ne doit pas pouvoir être employé de manière incohérente, c'est-à-dire avec deux types différents : voilà ce qu'il faut assurer par le typage. On connaît plusieurs systèmes de types qui atteignent ce résultat [64, 98, 21, 3, 40, 99] ; mais ils se révèlent trop restrictifs, rejetant en même temps que des programmes erronés de nombreux programmes corrects et d'un réel intérêt pratique. C'est pour pallier cette faiblesse que j'ai été amené à concevoir d'autres systèmes de types polymorphes pour les structures modifiables, dont la présentation et l'étude constituent l'essentiel de cette Thèse.

Pour illustrer ces généralités, prenons comme exemple la plus simple des restrictions qui interdisent les utilisations incohérentes d'un objet mutable : elle consiste à exiger que tous les objets modifiables soient créés avec des types monomorphes clos, c'est-à-dire ne contenant aucune variable de types. C'est la solution retenue dans les premières implémentations de ML, comme par exemple le système Caml [19, 98]. Le système de types ainsi obtenu assure de manière évidente la sûreté de l'exécution. Malheureusement, il ne permet pas d'écrire des fonctions génériques qui créent des structures modifiables. C'est un inconvénient majeur en pratique, comme le montre l'exemple qui suit. Supposons défini un type abstrait τ `matrix` des matrices d'éléments de type τ, modifiables en place. La fonction suivante est d'intérêt général :

[1] Les exemples de cette Introduction supposent que le lecteur a quelques notions de ML. J'utilise la syntaxe du dialecte CAML [19, 57] ; les habitués de SML ajouteront `val` et `end` après `let`. Voici une trousse de survie pour le lecteur qui ignore tout de ML. La plupart des constructions syntaxiques se lisent comme de l'anglais. [] représente la liste vide, $[a]$ la liste à un élément a, et $[a_1;\ldots;a_n]$ la liste à n éléments $a_1 \ldots a_n$. Le type τ `list` est le type des listes dont les éléments sont de type τ ; la liste vide [] est du type τ `list` pour tout type τ. Les références sont des cellules d'indirections modifiables en place — en d'autres termes, des tableaux de taille 1. `ref`(a) alloue une nouvelle référence, initialisée à la valeur a. $!r$ renvoie le contenu courant de la référence r. $r := a$ remplace par a le contenu de la référence r.

```
let transpose_matrix m1 =
    let m2 = new_matrix(dim_y(m1), dim_x(m1), matrix_elt(m1,0,0)) in
    for x = 0 to dim_x(m1) - 1 do
      for y = 0 to dim_y(m1) - 1 do
        set_matrix_elt(m2, x, y, matrix_elt(m1,y,x))
      done
    done;
    m2
```

Clairement, cette fonction peut s'appliquer à des matrices de n'importe quel type : son type naturel est donc α `matrix` $\rightarrow$ α `matrix` pour tout type α. Pourtant, le système de types de Caml ne permet pas de lui donner ce type. La fonction ci-dessus crée en effet une matrice `m2` dont le type (α `matrix`) n'est pas clos. Pour que la fonction `transpose_matrix` soit bien typée en Caml, il faut, par une contrainte de types, restreindre le paramètre `m1` à un type monomorphe particulier, par exemple `int matrix`. On obtient ainsi une fonction qui transpose uniquement les matrices d'entiers. Pour transposer une matrice de nombres flottants, il faut réécrire cette fonction une deuxième fois — exactement comme si on programmait dans un langage monomorphe comme Pascal. On ne peut donc pas fournir de bibliothèque de fonctions génériques sur les matrices modifiables en place. Il en va de même pour de nombreuses structures de données bien connues (tables de hachage, graphes, arbres binaires équilibrés, *B-trees*), qui, pour être efficaces, doivent être implémentées à l'aide de modifications en place de la structure : on ne peut pas fournir une bibliothèque de fonctions génériques qui implémentent une fois pour toutes des tables de hachages ou des *B-trees*.

Pour résoudre ce problème, on a proposé d'autres systèmes de types pour les objets modifiables, plus fins mais aussi plus complexes que celui que j'ai pris comme exemple jusqu'ici. Ces systèmes plus fins typent de manière plus satisfaisante les fonctions génériques sur les structures de données modifiables. Un exemple bien connu est le système du Standard ML [64, 63, 71], qui utilise la notion de "variables impératives" [91, 92], et une extension simple de ce système, les "variables faibles", qui est utilisée par le compilateur Standard ML of New Jersey [3]. Ce système donne aux fonctions génériques sur les matrices, les tables de hachage, etc., des types polymorphes légèrement restreints, appelés types faiblement polymorphes, qui se révèlent en pratique suffisamment généraux pour permettre la réutilisation de ces fonctions. Cependant, des insuffisances graves de ce système apparaissent lorsqu'on se met à construire d'autres fonctions génériques au-dessus de ces fonctions faiblement polymorphes.

Première insuffisance : les fonctions faiblement polymorphes interagissent mal avec la pleine fonctionnalité. Par exemple, les deux fragments de code ci-dessous ne sont pas équivalents :

```
let makeref1 = function x → ref x in ...
let makeref2 = (function f → f)(function x → ref x) in ...
```

La fonction `makeref1` est faiblement polymorphe ; la fonction `makeref2`, complètement monomorphe. Insérer une application de l'identité change donc le typage d'un programme ; c'est bien étrange. D'autres bizarreries apparaissent lorsqu'on applique partiellement des fonctionnelles à des fonctions faiblement polymorphes :

```
let mapref1 = map (function x → ref x) in ...
let mapref2 = function l → map (function x → ref x) l in ...
```

La fonction `mapref1` est monomorphe ; la fonction `mapref2`, faiblement polymorphe. Le typage de SML n'est donc pas stable par eta-réduction ; c'est, encore une fois, bien étrange[2]. Il faut noter que les deux bizarreries mentionnées, bien qu'elles suffisent sans doute à disqualifier le système de types de Standard ML auprès des théoriciens, ne font pas vraiment obstacle à la programmation : on peut les contourner en écrivant différemment les programmes.

Ce n'est pas le cas de la deuxième insuffisance majeure du typage des constructions impératives en Standard ML : les fonctions génériques qui utilisent des structures modifiables de manière interne, comme intermédiaires de calcul, n'ont pas des types complètement polymorphes. Il n'y a pas de moyen simple de contourner cette deuxième insuffisance. C'est grave ; car bien souvent l'emploi interne de structures modifiables est indispensable pour implémenter efficacement des fonctions génériques. Exemple : les algorithmes classiques sur les graphes (parcours en profondeur ou en largeur, calcul des composantes connexes, tri topologique, etc.) commencent très souvent par allouer localement un tableau, une pile ou une file d'attente de sommets [87]. Voici un exemple de fonction générique écrite dans ce style :

```
let reverse1 l =
    let arg = ref l in
    let res = ref [] in
    while not is_null(!arg) do
        res := cons(head(!arg), !res);
        arg := tail(!arg)
    done;
    !res
```

Cette fonction renverse une liste. Elle utilise, de manière purement interne, deux références. Voici une autre fonction qui elle aussi renverse une liste, sans employer de références cette fois.

```
let reverse2 l =
    let rec rev arg res =
        if null(arg) then res else rev (cons(head(arg), res)) (tail(res))
    in rev l []
```

On peut préférer l'une à l'autre, par des considérations de style ou d'efficacité. Il n'en reste pas moins que ces deux fonctions sont, vues de l'extérieur, exactement équivalentes. Et pourtant, le système de types de Standard ML donne à `reverse1` un type moins général qu'à `reverse2` : le type de `reverse1` est faiblement polymorphe, interdisant d'appliquer `reverse1` à la liste vide polymorphe, alors que le type de `reverse2` est complètement polymorphe, et permet d'appliquer `reverse2` à n'importe quelle liste. Toute fonction polymorphe qui utilise `reverse1`, directement ou indirectement, va recevoir un type faiblement polymorphe, moins général que si elle utilisait `reverse2`. On ne peut donc pas, dans une bibliothèque sur les listes, remplacer `reverse2` par `reverse1`, bien que ces deux fonctions aient exactement la même sémantique. Ceci va à l'encontre des excellents principes de programmation modulaire [70] que le standard ML tente précisément

[2]Dans le système de types du noyau ML, comme dans la plupart des systèmes de types connus, si une expression a a le type τ sous certaines hypothèses, et si a s'eta-réduit en a', alors a' a également le type τ. Ce n'est pas le cas en SML, puisque `mapref2` s'eta-réduit en `mapref1`.

de promouvoir [54] : un détail d'implémentation, le style de programmation (impératif ou bien purement applicatif), interfère avec une des spécifications, le type.

On l'a vu, le standard ML et ses extensions simples n'ont guère réussi dans leur tentative de combiner typage polymorphe et constructions impératives : ils permettent de programmer ou bien de manière générique, ou bien dans le style impératif, mais pas les deux à la fois. Il est difficile de discuter l'importance de ce fait sans tomber dans la polémique entre partisans de la programmation purement applicative ("il n'y a que ça de propre") et défenseurs du style impératif ("il n'y a que ça qui marche pour les vrais programmes"). J'adopte ici le point de vue d'un programmeur qui cherche un langage algorithmique — un langage permettant l'expression directe des algorithmes connus — pour la programmation à grande échelle. La plupart des algorithmes sont décrits dans la littérature sous forme de pseudo-code, mélangeant style déclaratif ("soit x un sommet du graphe de degré minimal") et style impératif ("faire $E \leftarrow E \setminus \{x\}$ et recommencer jusqu'à $E = \emptyset$"). Traduire ces algorithmes dans un langage purement applicatif sans augmenter leur complexité nécessite souvent des trésors d'ingéniosité. (Le *Journal of Functional Programming* consacre une rubrique, *Functional pearls*, à ce genre de problèmes.) Nombreux sont les algorithmes efficaces dont on ne connaît pas encore de formulation purement applicative. Un énorme travail de traduction reste à accomplir avant qu'on puisse considérer les langages purement applicatifs comme des langages algorithmiques.

Pour ce qui est de la programmation à grande échelle, elle est facilitée par le découpage du programme en modules "autonomes" : des modules qui communiquent peu d'information entre eux, en suivant un protocole très strict. Avec des structures modifiables, chaque module peut contenir des variables d'état, et les tenir à jour lui-même, sans participation de ses clients. Dans un langage purement applicatif, ces informations d'état doivent être communiquées à l'extérieur et propagées par les clients du module, ce qui est source d'erreurs.

De ce point de vue, je conclus que les traits impératifs sont indispensables à la programmation algorithmique, dans l'état actuel des connaissances. Que le typage polymorphe de ces traits pose problème signifie donc qu'il reste à prouver qu'un langage algorithmique peut bénéficier des bienfaits du typage polymorphe. Un des buts de cette Thèse est de faire cette preuve, en proposant des systèmes de typages où le polymorphisme interagit beaucoup mieux avec le style impératif de programmation, sans pour autant compromettre la sûreté de l'exécution.

Les problèmes posés par le typage polymorphe des structures modifiables pourraient encore sembler anecdotiques s'ils n'étaient en rien spécifiques aux structures modifiables et au style impératif de programmation : des problèmes très similaires apparaissent lorsqu'on tente de typer avec polymorphisme un certain nombre d'autres traits de langages non purement applicatifs, provenant d'horizons très divers. J'en mentionnerai trois : les variables logiques, les canaux de communication, et les objets continuations.

Les variables logiques sont à la base de la plupart des langages de programmation dits logiques ; on les retrouve dans des propositions d'unification entre langages logiques et langages applicatifs [88, 74, 1]. Elles permettent de calculer sur des objets dont certaines parties sont initialement inconnues, et vont se trouver précisées par accumulation de contraintes durant l'exécution du programme. Grâce aux variables logiques, on peut éviter de formuler explicitement une stratégie de résolution de ces contraintes.

Les canaux de communication permettent les échanges de données entre programmes s'exécutant en parallèle. Ils sont à la base de nombreux calculs de processus communicants [59, 37]. (Je considère ici des calculs d'ordre supérieur, où les canaux sont eux-mêmes des valeurs "de première classe".) Cette extension des langages applicatifs permet la description d'algorithmes distribués. Aussi, certains problèmes en apparence séquentiels se résolvent plus élégamment sous forme de processus communicants.

Enfin, les objets continuations permettent aux programmes de capturer et de manipuler l'état courant de leur évaluation, et donc de définir leurs propres structures de contrôle, comme par exemple des mécanismes d'entrelacement des calculs (*coroutines*) ou des stratégies d'explorations (*non-blind backtracking*) taillés sur mesure pour le problème à résoudre. Les objets continuations se trouvent en Scheme [75], et dans certains systèmes ML [25].

J'aurais volontiers détaillé le typage des variables logiques, des canaux de communication et des continuations si je n'avais pas eu à répéter exactement la discussion des structures modifiables. Les trois extensions mentionnées peuvent être typées de la même manière qu'on type les références ou les tableaux. Le typage obtenu est sémantiquement correct en l'absence de polymorphisme ; il devient incorrect lorsqu'on ajoute le polymorphisme sans précautions supplémentaires. Dans le cas des variables logiques et des canaux, on se convainc facilement de ce fait sur des exemples très proches du premier exemple de cette Introduction. Tout comme les références, les variables logiques peuvent être modifiées en place — une seule fois, il est vrai — après leur création. Tout comme les références, les canaux de communication rendent possible la transmission de données entre un écrivain (une expression) et un lecteur (un contexte) qui ne sont pas adjacents dans le programme source. On retrouve donc immédiatement les mêmes problèmes que dans le cas des structures modifiables. Pour les continuations, il est plus difficile de faire le lien avec les structures modifiables : on a longtemps cru que le typage polymorphe simple des objets continuations était sûr (une "preuve" de cette affirmation a même été publiée), jusqu'à ce que des contre-exemples apparaissent. Les contre-exemples sont considérablement plus compliqués que ceux pour les références, mais reposent sur le fait que les objets continuations permettent de reprendre l'évaluation d'une expression dans un contexte qui ne correspond pas exactement au contexte dans lequel elle a été typée.

Nous sommes donc en présence de trois extensions utiles des langages purement applicatifs (variables logiques, canaux de communication et continuations) qui, bien que sans rapport direct avec le style impératif de programmation, posent les mêmes problèmes de typage polymorphe que les structures de données modifiables. On peut appliquer à ces trois extensions les systèmes de types connus pour les références. Par exemple, le système de Standard ML a été appliqué aux continuations [3, 100] et aux canaux de communication [80] ; le système de CAML, aux variables logiques [48]. Comme on pouvait s'y attendre, les insuffisances de ces systèmes se manifestent à nouveau, rendant difficile, voire impossible, l'écriture de fonctions génériques utilisant ou bien des variables logiques, ou bien des communications inter-processus, ou bien des captures de continuations. En conséquence, preuve n'a toujours pas été faite que le typage polymorphe pouvait s'appliquer avec profit 1- aux tentatives d'intégration logico-fonctionnelles fondées sur les variables logiques, 2- aux calculs de processus communicants avec les canaux comme valeurs, 3- aux manipulations de continuations comme valeurs. Voilà qui est trop, et c'est pourquoi je ne me limite pas dans cette Thèse à proposer des systèmes de types pour les structures mutables : j'applique aussi, avec succès, ces

systèmes aux canaux de communication et aux objets continuations.[3]

La thèse que je soutiens ici est que le typage polymorphe peut être appliqué de manière profitable à de nombreux traits de langages non purement applicatifs. Comme on va le voir dans la suite de ce travail, cette extension du domaine d'application du polymorphisme ne va pas sans efforts : se révèlent nécessaires des modifications d'une grande ampleur dans le système de types ou dans la sémantique de certaines constructions. Tel est, me semble-t'il, le prix à payer pour faire sortir le polymorphisme et ses bienfaits du ghetto des langages purement applicatifs.

Plan

Le reste de cette Thèse s'organise comme suit. Le premier chapitre présente le noyau du langage ML, sous la forme d'un petit langage applicatif muni du système de types polymorphe de Milner. Je formalise la sémantique et le typage de ce langage dans le cadre de la sémantique relationnelle, et je montre les propriétés habituelles de cohérence du typage vis-à-vis de l'évaluation, et d'existence d'un type principal.

Le chapitre 2 ajoute au langage applicatif du chapitre précédent trois sortes d'objets "de première classe" : les références, les canaux de communication et les continuations. Les références modélisent les structures de données modifiables en place ; les canaux, la communication de données entre processus ; les continuations, les structures de contrôle non locales. Je donne les nouvelles règles d'évaluation pour chacune de ces trois extensions, et je montre que leurs règles de typage naturelles ne sont pas correctes vis-à-vis de ces règles d'évaluation, en raison de la présence de types polymorphes.

Le chapitre 3 présente un nouveau système de types polymorphes pour les références, les canaux et les continuations. Ce système, bien que proche du système de Milner, s'en distingue par des restrictions supplémentaires sur l'étape de généralisation (on s'interdit de généraliser certaines variables de types : les variables en position dangereuse), d'une part, et de l'autre par un typage plus fin des valeurs fonctionnelles : le typage des fermetures. Je commence par présenter informellement ce système, et montrer qu'il est bien adapté à la programmation dans le style impératif. Je formalise ensuite les règles de typage, et prouve qu'elles sont cohérentes avec les règles d'évaluation pour les références, les canaux et les continuations. Enfin, je montre la propriété de type principal pour ce système.

Dans le chapitre 4, je montre que le système du précédent chapitre n'est pas entièrement satisfaisant, car il rejette certains programmes "purs" bien typés en ML, et j'en propose une variante qui, bien que fondée sur les mêmes idées, ne présente pas cet inconvénient. Cette variante nécessite un appareillage technique nettement plus lourd que celui employé jusqu'ici, et la présenter directement, comme dans [51], n'est guère pédagogique ; c'est pourquoi j'ai préféré présenter d'abord le système du chapitre 3, même s'il est dépassé par le système du chapitre 4. Je démontre à nouveau

[3] L'application aux variables logiques a été faite par Vincent Poirriez [74], pour une première version d'un de mes systèmes. Dans la présente Thèse, structures modifiables, canaux et continuations sont étudiés indépendamment les uns des autres. Je ne vois aucune difficulté à combiner structures modifiables et continuations, comme dans le langage Scheme. Il est difficile d'attribuer une sémantique sensée aux autres combinaisons (structures modifiables et canaux, continuations et canaux).

les propriétés de type principal et de cohérence vis-à-vis des règles d'évaluation, et je montre que ce système est bien une extension du système de ML.

Le chapitre 5 tente d'évaluer l'apport de ce travail d'un point de vue pratique. Je compare les systèmes de types que j'ai proposés avec les autres systèmes publiés dans la littérature, d'une part sous l'angle de l'expressivité (tel programme utile et correct est-il reconnu bien typé ?), et d'autre part sous l'angle de la compatibilité avec la programmation modulaire (est-il difficile de spécifier les types des fonctions exportées par un module ?)

Le chapitre 6 montre que les difficultés rencontrées dans le typage polymorphe des références, canaux et continuations sont essentiellement liées à la sémantique qu'attribue ML à la généralisation et à la spécialisation (les deux constructions introduisant le polymorphisme), et que j'appelle la sémantique "par valeur" du polymorphisme : si on donne au polymorphisme une autre sémantique, la sémantique "par nom", les règles de typage naïves pour les références, les canaux et les continuations se révèlent correctes. C'est le principal résultat de ce chapitre, et il suggère qu'un langage avec polymorphisme par nom pourrait être une alternative intéressante au langage ML.

Lectures

Voici quelques itinéraires possibles à travers cette Thèse. Tous présupposent une certaine connaissance du langage ML, telle qu'on peut l'acquérir par la lecture des premiers chapitres de [71, 57] ou de [34, 19], et des idées de la sémantique relationnelle [73, 42]. Le lecteur qui désire comprendre le problème peut se contenter de lire le chapitre 1 (surtout les parties 1.1 et 1.3) et le chapitre 2 (on peut survoler les parties 2.1.2, 2.2.2 et 2.3.2). Le lecteur qui veut aussi avoir une idée des solutions que je propose y ajoutera les parties 3.1 et 4.1, le chapitre 5, et la partie 6.1. Ceux qui veulent voir des preuves de cohérence et de principalité devront lire les chapitres 1, 2, 3 et 6 en entier. La lecture complète du chapitre 4 n'est recommandée qu'aux experts.

Conventions

Cette Thèse est écrite en français moderne ; et donc, l'auteur emploie la première personne du singulier pour parler de ses opinions, points de vue, choix méthodologiques et résultats obtenus, toutes choses fort personnelles qu'il ne convient pas de noyer dans l'habituel "nous". La première personne du pluriel désigne l'auteur et le lecteur, unis main dans la main à des fins didactiques. J'emploie largement le pronom "on" pour énoncer des vérités générales, ou du moins ce que je crois être telles.

Pour ne pas alourdir l'exposition, les définitions formelles sont données au fil du texte, et non composées à part comme les énoncés de propositions. Je les signale cependant en mettant en PETITES CAPITALES le mot qu'on est en train de définir précisément. Toutes ces définitions sont indexées à la fin du volume.

Certains paragraphes sont marqués "Contexte". Ils décrivent brièvement d'autres approches et situent mon travail parmi elles. Ceci fournit au lecteur cultivé des points de repère, et au lecteur curieux des pointeurs vers la littérature ; le lecteur tendance Bourbaki les tiendra pour inutiles.

Chapitre 1

Un langage applicatif polymorphe

Le langage ML tout entier est trop complexe pour qu'on puisse raisonner formellement dessus sans trop de peine. Ce chapitre introduit un petit langage purement applicatif avec typage polymorphe, qui présente les traits principaux de ML, tout en étant beaucoup plus simple. Ce langage est la base des extensions que l'on étudie dans les chapitres suivants.

Le présent chapitre ne contient rien qui n'ait pas été déjà écrit de nombreuses fois par ailleurs [58, 22, 21, 91, 17, 64, 63, 92]. Je m'efforce cependant de donner une présentation complète et nécessitant peu de connaissances préalables. C'est l'occasion de fixer la plupart des notations, des techniques de définition, et des techniques de preuve qu'on utilise abondamment par la suite.

1.1 Syntaxe

Les EXPRESSIONS du langage (notées a, éventuellement avec un prime ou un indice) sont les éléments de l'algèbre de termes engendrée par la grammaire suivante :

a	$::=$	cst	constante
	$\mid$	x	identificateur
	$\mid$	$op(a)$	application de primitive
	$\mid$	f `where` $f(x) = a$	fonction (récursive)
	$\mid$	$a_1(a_2)$	application
	$\mid$	`let` $x = a_1$ `in` a_2	liaison `let`
	$\mid$	(a_1, a_2)	construction d'une paire

Dans cette grammaire, x et f décrivent un ensemble infini `Ident` d'IDENTIFICATEURS. Les expressions de la forme cst sont des CONSTANTES, prises dans un ensemble `Cst` qu'on laisse indéterminé, pour plus de généralité. Par exemple, `Cst` peut contenir des nombres entiers, les valeurs de vérité `true` et `false`, des chaînes de caractères, etc.

Dans les expressions de la forme $op(a)$, op est pris dans un ensemble d'OPÉRATEURS `Op` qu'on laisse lui aussi non spécifié. Typiquement, `Op` comprend les opérations arithmétiques usuelles sur

les entiers, les comparaisons entre entiers, les deux projections `fst` et `snd` pour les paires ; on y fera même rentrer des constructions conditionnelles comme `if`...`then`...`else`...

L'expression f `where` $f(x) = a$ définit la fonction de paramètre x et de résultat la valeur de a. L'identificateur f est le nom interne de la fonction. A l'intérieur de l'expression a, l'identificateur f est considéré lié à la fonction en train d'être définie. La fonction définie peut donc être récursive. Dans les exemples qui suivent, on note de manière plus classique $\lambda x.\, a$ les fonctions non récursives ; il est entendu que $\lambda x.\, a$ est une abréviation pour f `where` $f(x) = a$, où f est un identificateur quelconque n'apparaissant pas dans a.

Contexte. Les petits langages qu'on considère habituellement ne comprennent pas de fonctions récursives. En conséquence, ils manquent désagréablement de réalisme : on ne peut guère y écrire d'exemples intéressants. Aussi, ils ont souvent des propriétés qui sont fausses dans tous les vrais langages de programmation : "tout programme bien typé termine", par exemple. De plus, il est difficile d'ajouter la récursion après coup : les preuves, et même parfois la formalisation, doivent être entièrement revues. J'ai donc choisi de considérer dès le début un langage récursif.

Dans le langage ML, la récursion se présente comme une extension de la construction `let` : `let rec`, `let val rec`, `let fun` suivant les syntaxes. J'ai préféré présenter la récursion comme une extension de la λ-abstraction ; de la sorte, on montre clairement que seules les fonctions peuvent être définies récursivement, et non pas n'importe quelle valeur. (En ML, il faut des restrictions supplémentaires pour interdire des définitions comme `let rec x` $= \mathtt{x} + 1$, qu'on ne sait pas exécuter.) En syntaxe Standard ML, f `where` $f(x) = a$ s'écrit `let fun` $f\ x = a$ `in` f `end` ; en syntaxe Caml, `let rec` $f\ x = a$ `in` f, ou encore f `where rec` $f\ x = a$. □

Exemple. Voici deux expressions syntaxiquement correctes du langage :

$$\mathtt{let\ fact} = \mathtt{f\ where\ f(n)} = \mathtt{ifthenelse}(\leq(\mathtt{n}, 0), (1, \times(\mathtt{n}, \mathtt{f}(-(\mathtt{n}, 1))))) \ \mathtt{in\ fact}(2)$$

$$\mathtt{let\ double} = \lambda\mathtt{f}.\, \lambda\mathtt{x}.\, \mathtt{f}(\mathtt{f}(\mathtt{x}))\ \mathtt{in\ double}(\lambda\mathtt{x}.\ +(\mathtt{x}, \mathtt{true}))(1)$$

Pour améliorer la lisibilité des exemples, on se permet d'utiliser les notations infixes usuelles pour les opérateurs binaires et ternaires, ainsi que les règles de priorité et d'associativité habituelles. On préfère donc écrire :

$$\mathtt{let\ fact} = \mathtt{f\ where\ f(n)} = \mathtt{if\ n} \leq 0\ \mathtt{then\ 1\ else\ n} \times \mathtt{f(n} - 1)\ \mathtt{in\ fact}(2)$$

$$\mathtt{let\ double} = \lambda\mathtt{f}.\, \lambda\mathtt{x}.\, \mathtt{f}(\mathtt{f}(\mathtt{x}))\ \mathtt{in\ double}(\lambda\mathtt{x}.\ \mathtt{x} + \mathtt{true}))(1)$$

□

On note $\mathcal{I}(a)$ l'ensemble des IDENTIFICATEURS LIBRES dans l'expression a. Les constructions liantes sont le `let` et le `where`. La définition exacte de $\mathcal{I}$ est :

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{I}(cst) & = & \emptyset & \mathcal{I}(x) & = & \{x\} \\
\mathcal{I}(op(a)) & = & \mathcal{I}(a) & \mathcal{I}(f\ \mathtt{where}\ f(x) = a) & = & \mathcal{I}(a) \setminus \{f, x\} \\
\mathcal{I}(a_1(a_2)) & = & \mathcal{I}(a_1) \cup \mathcal{I}(a_2) & \mathcal{I}(\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2) & = & \mathcal{I}(a_1) \cup (\mathcal{I}(a_2) \setminus \{x\}) \\
\mathcal{I}(a_1, a_2) & = & \mathcal{I}(a_1) \cup \mathcal{I}(a_2) & & &
\end{array}$$

1.2 Sémantique

On va maintenant donner une signification aux expressions du langage, en définissant un procédé d'évaluation qui, à chaque expression, associe un résultat d'évaluation. On utilise pour ce faire le formalisme de la sémantique relationnelle, qui consiste à définir par un système de règles d'inférences un prédicat entre expressions et résultats, le JUGEMENT D'ÉVALUATION, disant si oui ou non tel terme peut s'évaluer en tel résultat.

Contexte. Cette méthode est connue sous le nom de "sémantique opérationnelle structurée" (SOS) dans le Nord de la Grande-Bretagne [73], et de "sémantique naturelle" dans le Midi de la France [42]. Au contraire des techniques dénotationnelles, la sémantique relationnelle ne nécessite aucune structure mathématique compliquée ; elle est aussi plus proche d'une exécution directe sur machine [16]. J'aurais pu également procéder par réécriture du texte source (pour traiter les extensions du prochain chapitre, la réécriture simple ne suffit pas, il faut des règles avec contexte [26, 27, 100]), voire par traduction dans le code d'une machine abstraite [46, 18]. □

En pratique, le jugement d'évaluation fait intervenir des arguments supplémentaires, qui représentent le contexte dans lequel l'évaluation a lieu. Pour le langage considéré dans ce chapitre, on a besoin d'un seul argument supplémentaire : l'environnement d'évaluation, qui enregistre les liaisons de variables à des valeurs. Le jugement d'évaluation a donc la forme $e \vdash a \Rightarrow r$, qu'il faut lire : "dans l'environnement d'évaluation e, l'expression a s'évalue en le résultat r".

1.2.1 Objets sémantiques

Il est temps de définir précisément les différentes espèces d'objets qui interviennent dans la sémantique : les RÉSULTATS, les VALEURS et les ENVIRONNEMENTS D'ÉVALUATION. Ces objets sémantiques sont pris dans l'algèbre de termes engendrée par la grammaire suivante :

Résultats :	r	::=	v	résultat normal (une valeur)
		|	`err`	résultat d'erreur
Valeurs :	v	::=	cst	valeur de base
		|	(v_1, v_2)	paire de valeurs
		|	(f, x, a, e)	valeur fonctionnelle (fermeture)
Environnements :	e	::=	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	

Dans les expressions d'environnements e, on suppose les identificateurs $x_1 \ldots x_n$ tous différents. Ici et par la suite, le terme e de la forme $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ est interprété comme une application partielle à domaine finie des identificateurs dans les valeurs : l'application qui à x_i associe v_i, pour i de 1 à n, et qui n'est pas définie sur les autres identificateurs. On parle par abus de langage d'application finie des identificateurs dans les valeurs. L'application vide se note $[\,]$. On note $\mathrm{Dom}(e)$ l'ensemble de définition de e, c'est-à-dire $\{x_1, \ldots, x_n\}$, et $\mathrm{Im}(e)$ son ensemble image, c'est-à-dire $\{v_1, \ldots, v_n\}$. Si x appartient à $\mathrm{Dom}(e)$, on note $e(x)$ la valeur associée à x dans e. Enfin, on définit l'extension de e par v en x, notée $e + x \mapsto v$, par :

$$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] + x \mapsto v \;=\; [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, x \mapsto v] \quad \text{si } x \notin \{x_1, \ldots, x_n\}$$

$$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] + x \mapsto v \quad = \quad [x_1 \mapsto v_1, \ldots, x_{i-1} \mapsto v_{i-1}, x \mapsto v, x_{i+1} \mapsto v_{i+1}, \ldots, x_n \mapsto v_n] \quad \text{si } x = x_i$$

On a donc, de manière très naturelle,

$$\begin{aligned} \mathrm{Dom}(e + x \mapsto v) &= \mathrm{Dom}(e) \cup \{x\} \\ (e + x \mapsto v)(x) &= x \\ (e + x \mapsto v)(y) &= e(y) \text{ pour tout } y \in \mathrm{Dom}(e),\ y \neq x \end{aligned}$$

1.2.2 Règles d'évaluation

Ces définitions étant posées, nous sommes maintenant en mesure de définir le jugement d'évaluation $e \vdash a \Rightarrow r$. La définition se présente comme un ensemble d'axiomes et de règles d'inférence. Mode d'emploi : un axiome P permet de conclure que la proposition P est vraie ; une règle d'inférence de la forme

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{P}$$

permet d'établir que la conclusion P est vraie à partir du moment où toutes les prémisses $P_1 \ldots P_n$ sont établies. Les axiomes et les règles peuvent contenir des (méta-) variables. Ces variables sont de façon implicite quantifiées universellement en tête de chaque règle.

$$e \vdash cst \Rightarrow cst \qquad \frac{x \in \mathrm{Dom}(e)}{e \vdash x \Rightarrow e(x)} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x \Rightarrow \texttt{err}}$$

L'axiome de gauche dit qu'une constante s'évalue en elle-même. La première règle dit qu'une variable s'évalue en la valeur qui lui est attribuée par l'environnement e, si cette variable est bien dans le domaine de e. Dans le cas contraire, il n'y a pas de valeur sensée qu'on puisse attribuer à l'expression x. La deuxième règle fait donc renvoyer dans ce cas la réponse "il s'est produit une erreur", notée `err`.

$$e \vdash (f \ \texttt{where} \ f(x) = a) \Rightarrow (f, x, a, e)$$

Une fonction s'évalue en une FERMETURE : un objet qui associe le corps non évalué d'une fonction (le triplet f, x, a) avec l'environnement e présent au moment de la définition de la fonction.

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e \vdash a_2 \Rightarrow v_2}{e \vdash (a_1, a_2) \Rightarrow (v_1, v_2)} \qquad \frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_2 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}}$$

Une expression de paire s'évalue normalement en une paire de valeurs, de la manière évidente. Cependant, si l'évaluation d'une des deux composantes de la paire provoque une erreur (comme par exemple d'évaluer une variable x qui n'est pas liée dans e), alors l'évaluation de la paire provoque elle aussi une erreur.

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e + x \mapsto v_1 \vdash a_2 \Rightarrow r_2}{e \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 \Rightarrow r_2} \qquad \frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 \Rightarrow \texttt{err}}$$

Une liaison `let` évalue d'abord son premier argument, associe dans l'environnement la valeur obtenue à la variable liée, et évalue son deuxième argument dans l'environnement ainsi augmenté. Le résultat obtenu est aussi le résultat de l'évaluation de l'expression `let` tout entière. Dans le cas où l'évaluation du premier argument provoque une erreur, l'évaluation de l'expression `let` s'arrête immédiatement sur une erreur.

$$\frac{e \vdash a_1 \Rightarrow (f, x, a_0, e_0) \qquad e \vdash a_2 \Rightarrow v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \Rightarrow r_0}{e \vdash a_1(a_2) \Rightarrow r_0}$$

La règle pour l'application $a_1(a_2)$ est la plus complexe de toutes. L'expression a_1 doit s'évaluer en une fermeture (f, x, a_0, e_0). L'argument a_2 doit s'évaluer en une valeur v_2. On évalue alors le corps de fonction a_0 provenant de la fermeture, dans l'environnement e_0 provenant lui aussi de la fermeture, après avoir enrichi e_0 de deux liaisons : le paramètre de fonction x est lié à la valeur v_2 de l'argument ; le nom f de la fonction est lié à la fermeture elle-même. Cette dernière liaison assure que l'évaluation de a_0, qui peut faire appel à f, trouvera dans l'environnement la valeur correcte de f.

$$\frac{e \vdash a_1 \Rightarrow r_1 \qquad r_1 \text{ n'est pas de la forme } (f, x, a_0, e_0)}{e \vdash a_1(a_2) \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_2 \Rightarrow \texttt{err}}{e \vdash a_1(a_2) \Rightarrow \texttt{err}}$$

Voici les deux cas d'erreur pour l'application de fonction. L'application $a_1(a_2)$ est absurde si a_1 s'évalue en autre chose qu'une fermeture. Exemple : 1(2). La première règle renvoie donc `err` dans ce cas. La deuxième règle assure que la réponse `err` se propage à travers les applications : si l'évaluation de la partie argument stoppe sur une erreur, l'évaluation de l'application tout entière en fait autant (même si la fonction n'utilise pas son argument).

Il reste à donner des règles d'évaluation pour chacune des primitives de l'ensemble `Op`. A titre d'exemple, voici les règles pour l'addition entière, et pour la conditionnelle :

$$\frac{e \vdash a \Rightarrow (cst_1, cst_2) \qquad cst_1 \in \texttt{Int} \qquad cst_2 \in \texttt{Int} \qquad cst_1 + cst_2 = cst}{e \vdash +(a) \Rightarrow cst}$$

$$\frac{e \vdash a \Rightarrow r \qquad r \text{ n'est pas de la forme } (cst_1, cst_2) \text{ avec } cst_1 \in \texttt{Int} \text{ et } cst_2 \in \texttt{Int}}{e \vdash +(a) \Rightarrow \texttt{err}}$$

$$\frac{e \vdash a_1 \Rightarrow \texttt{true} \qquad e \vdash a_2 \Rightarrow r_2}{e \vdash \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3 \Rightarrow r_2} \qquad \frac{e \vdash a_1 \Rightarrow \texttt{false} \qquad e \vdash a_3 \Rightarrow r_3}{e \vdash \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3 \Rightarrow r_3}$$

$$\frac{e \vdash a_1 \Rightarrow r_1 \qquad r_1 \notin \texttt{Bool}}{e \vdash \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3 \Rightarrow \texttt{err}}$$

Ici comme par la suite, on associe à ces règles d'inférence une relation entre environnements, expressions et valeurs, qu'on note encore $e \vdash a \Rightarrow v$, et qui est la relation la moins définie qui vérifie les règles d'inférences. Par la suite, on utilise très souvent la caractérisation suivante de cette relation : (e, a, v) sont en relation par $\cdot \vdash \cdot \Rightarrow \cdot$ si et seulement si il existe une dérivation finie du jugement $e \vdash a : v$ à partir des axiomes, par application répétée des règles d'inférence [73].

Contexte. L'introduction de la réponse `err` et des règles d'évaluation qui la font apparaître (les "règles d'erreurs") est la solution traditionnelle au problème de distinguer les programmes erronés des programmes qui ne terminent pas : sans règles d'erreurs, on ne sait pas, lorsqu'une expression n'admet pas de dérivation (finie) d'évaluation, si c'est parce qu'à un certain point aucune règle d'évaluation "normale" ne s'applique (cas d'un programme mal typé), ou si c'est parce que les seules dérivations possibles sont infinies (cas d'un programme qui ne termine pas).

Outre d'alourdir la définition de la sémantique, les règles d'erreurs présentent l'inconvénient supplémentaire d'imposer partiellement la stratégie d'évaluation — stratégie qui ne devrait pas entrer en ligne de compte à ce niveau de description. Par exemple, les deux règles d'erreurs pour la paire proposées ci-dessus :

$$\frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_2 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}}$$

impliquent une évaluation en parallèle des deux composantes de la paire ("non-déterminisme angélique"), puisque il faut renvoyer une erreur dans tous les cas où l'une ne termine pas alors que l'autre déclenche une erreur. Pour se laisser la possibilité d'une implémentation séquentielle, il faut dissymétriser les règles et prendre par exemple :

$$\frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_1 \Rightarrow v_1 \qquad e \vdash a_2 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}}$$

Mais, du coup, on a imposé l'ordre d'évaluation de la paire : de gauche à droite. D'autres approches plus récentes, qui se passent des règles d'erreurs, permettent d'éviter ce problème. Ces approches passent par des notions plus riches de dérivation : les dérivation partielles [32], ou les dérivations éventuellement infinies [20].

Ce problème de la spécification de l'ordre d'évaluation par les règles d'erreurs est moins important pour les langages statiquement typés : quand on n'évalue que des termes bien typés, les cas d'erreurs ne se produisent jamais à l'exécution, et donc on peut adopter une stratégie d'évaluation qui ne respecte pas les règles d'erreurs. C'est pourquoi je me suis permis d'employer, pour la paire et pour l'application de fonction, des règles d'erreurs non séquentielles : ce sont les plus simples à exprimer. □

1.3 Typage

On va maintenant munir le langage de règles de typage. Les règles de typage associent aux expressions des types (ou, plus exactement, des expressions de types), de la même manière que les règles d'évaluation leur associent des résultats d'évaluation. On emploie le système de types de Milner [58, 22], qui est à la base des systèmes de types du langage ML et de langages fonctionnels proches (Miranda, Hope, Haskell). Le trait distinctif du système de Milner est le polymorphisme : ce système prend en compte le fait qu'une même expression peut appartenir à plusieurs types différents.

1.3.1 Types

On se donne un ensemble `TypBas` de TYPES DE BASE (par exemple, `int`, le type des entiers, et `bool`, le type des valeurs de vérité), et un ensemble infini `VarTyp` de VARIABLES DE TYPES.

$$\begin{array}{rcll} \iota & \in & \texttt{TypBas} = \{\texttt{int};\texttt{bool};\ldots\} & \text{types de base} \\ \alpha,\beta,\gamma & \in & \texttt{VarTyp} & \text{variables de type} \end{array}$$

On définit l'ensemble `Typ` des EXPRESSIONS DE TYPES (ou simplement des TYPES), notées τ, par la grammaire suivante :

$$\begin{array}{rcll} \tau & ::= & \iota & \text{type de base} \\ & | & \alpha & \text{variable de type} \\ & | & \tau_1 \to \tau_2 & \text{type fonctionnel} \\ & | & \tau_1 \times \tau_2 & \text{type produit} \end{array}$$

On note $\mathcal{L}(\tau)$ l'ensemble des variables de types libres dans un type τ. Voici la définition exacte :

$$\begin{array}{rcl} \mathcal{L}(\iota) & = & \emptyset \\ \mathcal{L}(\alpha) & = & \{\alpha\} \\ \mathcal{L}(\tau_1 \to \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\ \mathcal{L}(\tau_1 \times \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \end{array}$$

1.3.2 Substitutions

Les substitutions considérées ici sont des applications finies des variables de types dans les expressions de types. On les note φ, ψ.

$$\text{Substitutions :} \quad \varphi,\psi \quad ::= \quad [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$$

Une substitution φ s'étend naturellement en un homomorphisme d'expressions de types, noté $\overline{\varphi}$, en définissant :

$$\begin{array}{rcl} \overline{\varphi}(\iota) & = & \iota \\ \overline{\varphi}(\alpha) & = & \varphi(\alpha) \text{ si } \alpha \in \mathrm{Dom}(\varphi) \\ \overline{\varphi}(\alpha) & = & \alpha \text{ si } \alpha \notin \mathrm{Dom}(\varphi) \\ \overline{\varphi}(\tau_1 \to \tau_2) & = & \overline{\varphi}(\tau_1) \to \overline{\varphi}(\tau_2) \\ \overline{\varphi}(\tau_1 \times \tau_2) & = & \overline{\varphi}(\tau_1) \times \overline{\varphi}(\tau_2) \end{array}$$

A partir de maintenant, on confond la substitution φ et son extension $\overline{\varphi}$, et on note φ pour les deux.

La propriété suivante montre l'effet d'une substitution sur les variables libres d'un type.

Proposition 1.1 (Substitution et variables libres) *Pour tout type τ et toute substitution φ, on a :*

$$\mathcal{L}(\varphi(\tau)) = \bigcup_{\alpha \in \mathcal{L}(\tau)} \mathcal{L}(\varphi(\alpha))$$

Démonstration : immédiat par récurrence structurelle sur τ. □

1.3.3 Schémas de types

On définit l'ensemble `SchTyp` des SCHÉMAS DE TYPES, notés σ, par la grammaire suivante :

$$\sigma ::= \forall \alpha_1 \dots \alpha_n . \tau \qquad \text{schéma de type}$$

Dans cette syntaxe, les variables quantifiées $\alpha_1 \dots \alpha_n$ sont considérées comme un ensemble de variables : leur ordre d'apparition n'est pas significatif, et on les suppose toutes différentes. On confond un type τ et le schéma de types trivial $\forall . \tau$, et on note τ pour les deux. On identifie deux schémas de types qui ne diffèrent que par un renommage des variables liées par $\forall$ (opération dite d'alpha-conversion), et par l'introduction ou la suppression de variables quantifiées non libres dans la partie type. Plus précisément, on quotiente l'ensemble des schémas par les deux équations suivantes :

$$\begin{aligned} \forall \alpha_1 \dots \alpha_n . \tau &= \forall \beta_1 \dots \beta_n . [\alpha_1 \mapsto \beta_1, \dots, \alpha_n \mapsto \beta_n](\tau) \\ \forall \alpha \alpha_1 \dots \alpha_n . \tau &= \forall \alpha_1 \dots \alpha_n . \tau \quad \text{si } \alpha \notin \mathcal{L}(\tau) \end{aligned}$$

Les variables libres d'un schéma de types sont :

$$\mathcal{L}(\forall \alpha_1 \dots \alpha_n . \tau) = \mathcal{L}(\tau) \setminus \{\alpha_1 \dots \alpha_n\}.$$

Cette définition passe bien au quotient par les deux équations ci-dessus, comme on le vérifie facilement à l'aide de la proposition 1.1.

On définit l'image d'un schéma par une substitution comme suit :

$$\varphi(\forall \alpha_1 \dots \alpha_n . \tau) = \forall \alpha_1 \dots \alpha_n . \varphi(\tau),$$

en supposant, après renommage des α_i si nécessaire, que les α_i sont HORS DE PORTÉE de φ ; c'est-à-dire, qu'aucun des α_i n'appartient au domaine de φ, et aucun des α_i n'est libre dans un des types de l'image de φ. On vérifie immédiatement que cette définition passe bien au quotient par les deux équations sur les schémas.

La proposition 1.1 est également vraie pour un schéma σ à la place du type τ.

1.3.4 Environnements de typage

Un environnement de typage, noté E, est une application finie des variables de programmes dans les schémas de types :

$$E \quad ::= \quad [x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n]$$

On définit l'image $\varphi(E)$ d'un environnement E par une substitution φ de la manière évidente :

$$\varphi([x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n]) = [x_1 \mapsto \varphi(\sigma_1), \ldots, x_n \mapsto \varphi(\sigma_n)].$$

On étend immédiatement l'opérateur $\mathcal{L}$ aux environnements de typage, en convenant qu'une variable de type est libre dans E si et seulement si elle est libre dans $E(x)$ pour un certain x :

$$\mathcal{L}(E) = \bigcup_{x \in \mathrm{Dom}(E)} \mathcal{L}(E(x))$$

1.3.5 Règles de typage

On va maintenant donner les règles de typage du langage, sous une forme très proche des règles d'évaluation du langage. Les règles d'inférence qui suivent définissent le JUGEMENT DE TYPAGE $E \vdash a : \tau$, qu'il faut lire : "dans l'environnement E, l'expression a possède le type τ". L'environnement E associe un schéma de types à chaque identificateur qui peut apparaître dans l'expression a.

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

La règle de typage des identificateurs dit qu'un identificateur x dans une expression peut être considérée avec n'importe quel type qui soit une instance du schéma de type associé à x par l'environnement de typage. On dit qu'un type τ est une INSTANCE d'un schéma de type σ de la forme $\forall \alpha_1 \ldots \alpha_n . \tau_0$, et on note $\tau \leq \sigma$, si et seulement si il existe une substitution φ de domaine inclus dans $\{\alpha_1 \ldots \alpha_n\}$ telle que τ est égal à $\varphi(\tau_0)$. La relation $\tau \leq \sigma$ passe bien au quotient par alpha-conversion et par élimination de variables quantifiées inutilisées.

$$\frac{E + f \mapsto (\tau_1 \to \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2}{E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \to \tau_2} \qquad \frac{E \vdash a_1 : \tau_2 \to \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1}$$

Une définition de fonction a le type $\tau_1 \to \tau_2$ à partir du moment où le corps de la fonction a le type τ_2, sous les hypothèses supplémentaires que son paramètre formel x a le type τ_1, et son nom interne f a le type $\tau_1 \to \tau_2$. Une application de fonction est bien typée dès lors que le type de l'argument correspond au type du paramètre de la fonction ; le type du résultat de la fonction est le type de l'application tout entière.

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let} \ x = a_1 \ \texttt{in} \ a_2 : \tau_2}$$

La construction `let` est la seule qui introduit dans l'environnement des identificateurs avec des types polymorphes (c'est-à-dire, des schémas de types non triviaux). Ceci est dû à l'action de l'opérateur de généralisation `Gen`, qui construit un schéma de type à partir d'un type et d'un environnement de typage, comme suit :

$$\texttt{Gen}(\tau, E) = \forall \alpha_1 \dots \alpha_n . \tau \quad \text{avec} \quad \{\alpha_1 \dots \alpha_n\} = \mathcal{L}(\tau) \setminus \mathcal{L}(E).$$

On verra plus bas que si a_1 appartient au type τ sous les hypothèses E, et si α n'est pas libre dans E, alors a_1 appartient aussi au type $[\alpha \mapsto \tau_1](\tau)$ pour tout type τ_1. Ceci justifie intuitivement qu'on quantifie universellement la variable α, puisqu'on peut la substituer indifféremment par n'importe quel type.

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2} \qquad \frac{\tau \leq \texttt{TypCst}(cst)}{E \vdash cst : \tau} \qquad \frac{\tau_1 \to \tau_2 \leq \texttt{TypOp}(op) \qquad E \vdash a : \tau_1}{E \vdash op(a) : \tau_2}$$

Le typage de la paire se passe de commentaires. Pour les constantes et les primitives, on suppose données deux fonctions, $\texttt{TypCst} : \texttt{Cst} \to \texttt{SchTyp}$ et $\texttt{TypOp} : \texttt{Op} \to \texttt{SchTyp}$, qui associent des schémas de types aux constantes et aux opérateurs. Par exemple, on peut prendre :

$$\begin{array}{rcll} \texttt{TypCst}(i) & = & \texttt{int} & (i = 0, 1, \dots) \\ \texttt{TypCst}(b) & = & \texttt{bool} & (b = \texttt{true} \text{ ou } \texttt{false}) \\ \texttt{TypOp}(+) & = & \texttt{int} \times \texttt{int} \to \texttt{int} & \\ \texttt{TypOp}(\texttt{ifthenelse}) & = & \forall \alpha.\ \texttt{bool} \times \alpha \times \alpha \to \alpha & \end{array}$$

Exemple. L'expression $\texttt{let id} = (\texttt{f where f(x)} = \texttt{x})\ \texttt{in id(1)}$ est du type `int`. En effet, la dérivation suivante est valide :

$$\dfrac{\dfrac{\dfrac{t \leq t}{[\texttt{f} \mapsto t \to t, \texttt{x} \mapsto t] \vdash \texttt{x} : t}}{[\,] \vdash \texttt{f where f(x)} = \texttt{x} : t \to t} \qquad \dfrac{\dfrac{\texttt{int} \to \texttt{int} \leq \forall t.\, t \to t}{E \vdash \texttt{id} : \texttt{int} \to \texttt{int}} \qquad \dfrac{\texttt{int} \leq \texttt{int}}{E \vdash 1 : \texttt{int}}}{E = [\texttt{id} \mapsto \forall t.\, t \to t] \vdash \texttt{id(1)} : \texttt{int}}}{[\,] \vdash \texttt{let id} = (\texttt{f where f(x)} = \texttt{x})\ \texttt{in id(1)} : \texttt{int}}$$

□

Contexte. La présentation ci-dessus est essentiellement celle de la définition de Standard ML [64], et diffère légèrement de la présentation originelle de Damas et Milner [22]. Le système de Damas-Milner attribue des schémas de types aux expressions, non des types simples, et comporte deux règles séparées pour la généralisation et l'instanciation. Dans la présentation qu'on donne plus haut, ces deux opérations sont intégrées à la règle pour le `let` (cas de la généralisation), et aux règles pour les variables et les constantes (cas de l'instanciation). Les règles ci-dessus présentent l'avantage d'être définies par récurrence sur la syntaxe de l'expression (*syntax-directed*) : la forme de l'expression détermine la seule règle qui peut s'appliquer ; et les prémisses des règles font intervenir des sous-expressions strictes de l'expression dans la conclusion. Ceci facilite certaines preuves, en particulier celle de l'algorithme d'inférence de types. □

1.3.6 Propriétés des règles de typage

Comme on l'a laissé entendre en commentant la règle de typage du `let`, le jugement de typage est stable par substitution : si on peut prouver $E \vdash a : \tau$, on peut substituer dans E et dans τ n'importe quelles variables de types par des types quelconques ; le jugement obtenu est toujours prouvable.

Proposition 1.2 (Stabilité du typage par substitution) *Soient a une expression, τ un type, E un environnement de typage, et φ une substitution. Si $E \vdash a : \tau$, alors $\varphi(E) \vdash a : \varphi(\tau)$.*

Démonstration : la preuve est par récurrence structurelle sur a. On donne les deux seuls cas qui ne s'ensuivent pas immédiatement de l'hypothèse de récurrence.

• **Cas $a = x$.** On doit avoir $\tau \leq E(x)$, avec $E(x) = \forall \alpha_1 \ldots \alpha_n . \tau_0$. Après renommage si nécessaire, on peut supposer que les α_i sont hors de portée de φ. Soit ψ une substitution des α_i telle que $\tau = \psi(\tau_0)$. On définit une substitution θ de domaine $\{\alpha_1 \ldots \alpha_n\}$ par $\theta(\alpha_i) = \varphi(\psi(\alpha_i))$. On a :

$$\begin{array}{rcccll} \theta(\varphi(\alpha_i)) & = & \theta(\alpha_i) & = & \varphi(\psi(\alpha_i)) & \text{pour tout } i \\ \theta(\varphi(\beta)) & = & \varphi(\beta) & = & \varphi(\psi(\beta)) & \text{pour tout } \beta \text{ distinct des } \alpha_i \end{array}$$

D'où $\theta(\varphi(\tau_0)) = \varphi(\psi(\tau_0)) = \varphi(\tau)$, ce qui établit que $\varphi(\tau)$ est une instance de $\varphi(E(x))$. On peut donc dériver $\varphi(E) \vdash x : \varphi(\tau)$.

• **Cas $a = (\texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2)$.** La dérivation se termine par :

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2 : \tau}$$

Par définition, $\texttt{Gen}(\tau_1, E)$ est égal à $\forall \alpha_1 \ldots \alpha_n . \tau_1$, avec $\{\alpha_1 \ldots \alpha_n\} = \mathcal{L}(\tau_1) \setminus \mathcal{L}(E)$. Soient $\beta_1, \ldots, \beta_n$ des variables de type hors de portée de φ, et non libres dans E. On note ψ la substitution $\varphi \circ [\alpha_i \mapsto \beta_i]$. On a $\psi(E) = \varphi(E)$, puisque les variables α_i ne sont pas libres dans E.

On applique deux fois l'hypothèse de récurrence : à la prémisse de gauche, avec la substitution ψ ; et à la prémisse de droite, avec la substitution φ. Il vient des preuves de :

$$\psi(E) \vdash a_1 : \psi(\tau_1) \qquad\qquad \varphi(E) + x \mapsto \varphi(\texttt{Gen}(\tau_1, E)) \vdash a_2 : \varphi(\tau_2)$$

On calcule maintenant $\texttt{Gen}(\psi(\tau_1), \psi(E))$, à l'aide de la proposition 1.1.

$$\mathcal{L}(\psi(\tau_1)) \setminus \mathcal{L}(\psi(E)) = \left(\bigcup_{\alpha \in \mathcal{L}(\tau_1)} \mathcal{L}(\psi(\alpha)) \right) \setminus \left(\bigcup_{\alpha \in \mathcal{L}(E)} \mathcal{L}(\psi(\alpha)) \right)$$

Par construction de ψ, on a $\psi(\alpha_i) = \varphi(\beta_i) = \beta_i$. De plus, pour toute variable α qui n'est pas une des α_i, le terme $\psi(\alpha)$ est égal à $\varphi(\alpha)$ et ne contient aucune des β_i. Comme les α_i sont libres dans τ_1, mais pas libres dans E, on a donc :

$$\beta_i \in \bigcup_{\alpha \in \mathcal{L}(\tau_1)} \mathcal{L}(\psi(\alpha)) \qquad\qquad \beta_i \notin \bigcup_{\alpha \in \mathcal{L}(E)} \mathcal{L}(\psi(\alpha)).$$

D'où $\{\beta_1 \dots \beta_n\} \subseteq \mathcal{L}(\psi(\tau_1)) \backslash \mathcal{L}(\psi(E))$. On montre maintenant l'inclusion inverse. Soit $\beta \in \mathcal{L}(\psi(\tau_1))$, telle que β n'est pas une des β_i. Soit $\alpha \in \mathcal{L}(\tau_1)$ telle que $\beta \in \mathcal{L}(\psi(\alpha))$ (l'existence de α est assurée par la proposition 1.1). Nécessairement, α n'est pas une des α_i ; car sinon, β serait une des β_i. Donc $\alpha \in \mathcal{L}(E)$, d'où $\beta \in \bigcup_{\alpha \in \mathcal{L}(E)} \mathcal{L}(\psi(\alpha))$ et $\beta \notin \mathcal{L}(\psi(\tau_1)) \setminus \mathcal{L}(\psi(E))$. Par conséquent :

$$\texttt{Gen}(\psi(\tau_1), \psi(E)) = \forall \beta_1 \dots \beta_n . \psi(\tau_1) = \varphi(\forall \alpha_1 \dots \alpha_n . \tau) = \varphi(\texttt{Gen}(\tau_1, E))$$

par définition de l'image d'un schéma par une substitution. Comme $\psi(E)$ et $\varphi(E)$ sont identiques, les deux dérivations obtenues par récurrence établissent donc que :

$$\varphi(E) \vdash a_1 : \psi(\tau_1) \qquad \varphi(E) + x \mapsto \texttt{Gen}(\psi(\tau_1), \varphi(E)) \vdash a_2 : \varphi(\tau_2).$$

Appliquant la règle de typage du `let` à ces deux prémisses, on obtient le résultat recherché :

$$\varphi(E) \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \varphi(\tau_2).$$

□

Deuxième propriété des règles de typage, utile pour l'inférence de types : si on peut prouver $a : \tau$ sous les hypothèses E, alors on peut aussi le prouver sous des hypothèses plus fortes E'. Pour préciser cette notion de "plus fort", on dit qu'un schéma σ' est PLUS GÉNÉRAL qu'un schéma σ, et on note $\sigma' \geq \sigma$, si toute instance de σ est aussi instance de σ'.

Proposition 1.3 *Soient E, E' deux environnements de typage tels que $\mathrm{Dom}(E) = \mathrm{Dom}(E')$, et $E'(x) \geq E(x)$ pour tout $x \in \mathrm{Dom}(E)$. Si $E \vdash a : \tau$, alors $E' \vdash a : \tau$.*

Démonstration : récurrence facile sur la structure de a. Le cas de base $a = x$ est immédiat par hypothèse sur E et E'. Pour le cas $a = (\texttt{let } x = a_1 \texttt{ in } a_2)$, on montre $\mathcal{L}(E') \subseteq \mathcal{L}(E)$, ce qui entraîne $\texttt{Gen}(\tau_1, E') \geq \texttt{Gen}(\tau_1, E)$, où τ_1 est le type de a_1. On peut donc appliquer l'hypothèse de récurrence à la dérivation de typage de a_2 ; le résultat s'ensuit. □

1.4 Sûreté du typage

Un des buts majeurs du typage est d'exclure les programmes qui peuvent provoquer une erreur de type à l'évaluation. On a défini des règles de typage en toute indépendance des règles d'évaluation. Il reste donc à prouver que le but est atteint, c'est-à-dire que les règles de typage sont sûres (*sound*) vis-à-vis des règles d'évaluation. Le résultat principal de cette partie est le suivant : une expression close bien typée ne peut pas s'évaluer en `err`.

Proposition 1.4 (Sûreté faible) *Soient a une expression, τ un type, et r une réponse. Si on a $[] \vdash a : \tau$ et $[] \vdash a \Rightarrow r$, alors r n'est pas* `err`.

Contexte. Ce résultat n'affirme pas qu'il existe une valeur v en laquelle l'expression a s'évalue : un programme bien typé peut parfaitement ne jamais terminer. Le but du typage d'un langage de programmation n'est pas d'assurer la terminaison, mais de garantir la conformité structurelle des données. □

Il est difficile de prouver la proposition de correction faible directement, car elle ne se prête pas à la récurrence : pour montrer que $a_1(a_2)$ ne peut s'évaluer en `err`, il ne suffit pas de prouver que ni a_1, ni a_2 ne s'évaluent en `err` ; même dans ce cas, l'application $a_1(a_2)$ peut provoquer une erreur, si a_1 ne s'évalue pas en une fermeture, par exemple. On va donc prouver un résultat plus fort : l'évaluation d'un terme clos de type τ, si elle termine, non seulement ne termine pas sur `err`, mais encore produit une valeur qui, sémantiquement, appartient bien au type τ. Par exemple, une expression de type `int` ne peut s'évaluer qu'en une valeur entière.

1.4.1 Typage sémantique

Avant d'aller plus loin, il faut donc définir précisément cette notion sémantique d'appartenance d'une valeur v à un type τ. On définit les trois relations de typage sémantique que voici :

$\models v : \tau$	v est une valeur correcte du type τ
$\models v : \sigma$	v est une valeur correcte de toutes les instances du schéma σ
$\models e : E$	les valeurs contenues dans l'environnement d'évaluation e appartiennent bien aux schémas correspondants dans E

Voici leur définition précise :

- $\models cst : \texttt{int}$ si cst est un entier
- $\models cst : \texttt{bool}$ si cst est `true` ou `false`
- $\models (v_1, v_2) : \tau_1 \times \tau_2$ si $\models v_1 : \tau_1$ et $\models v_2 : \tau_2$
- $\models (f, x, a, e) : \tau_1 \to \tau_2$ s'il existe un environnement de typage E tel que
$$\models e : E \quad \text{et} \quad E \vdash (f \ \texttt{where}\ f(x) = a) : \tau_1 \to \tau_2$$
- $\models v : \sigma$ si pour tout type τ tel que $\tau \leq \sigma$, on a $\models v : \tau$
- $\models e : E$ si $\mathrm{Dom}(e) = \mathrm{Dom}(E)$, et pour tout $x \in \mathrm{Dom}(e)$, on a $\models e(x) : E(x)$.

Cette définition est bien fondée par récurrence structurelle sur la valeur. Dans tous les cas sauf celui des schémas de types, on définit $\models$ sur la valeur v en faisant appel à $\models$ uniquement sur des sous-termes stricts de v. Dans le cas d'un schéma de types, on fait appel à $\models$ pour la même valeur, mais pour un type simple ; la condition correspondante ne peut donc pas s'appliquer plusieurs fois de suite sans faire intervenir une autre condition, qui, elle, ne considère que des sous-termes stricts de v.

Contexte. Le recours au jugement de typage pour définir $\models$ sur les valeurs fonctionnelles est une technique due à Tofte [91, 92]. Une définition plus traditionnelle serait "une valeur fonctionnelle est du type $\tau_1 \to \tau_2$ si elle envoie toute valeur de type τ_1 sur une valeur de type τ_2". En termes plus formels : $\models (f, x, a, e) : \tau_1 \to \tau_2$ si, pour toute valeur v_1 et toute réponse r_2 telles que
$$\models v_1 : \tau_1 \quad \text{et} \quad e + f \mapsto (f, x, a, e) + x \mapsto v_1 \vdash a \Rightarrow r_2,$$
on a $r_2 \neq \texttt{err}$ et $\models r_2 : \tau_2$. C'est la condition de continuité qu'on trouve, par exemple, dans les modèles à base d'idéaux [55]. Cependant, une telle définition pose problème en présence de

fonctions récursives : la preuve de correction du typage par simple récurrence échoue, car, pour la règle de typage du `where`, on ne peut conclure $\models (f, x, a, e) : \tau_1 \to \tau_2$ sans avoir déjà établi $\models (f, x, a, e) : \tau_1 \to \tau_2$. Il semble nécessaire d'avoir recours à des principes de preuve plus complexes (co-induction [62]), ou à des structures mathématiques plus riches (domaines [33]). En faisant appel au jugement de typage pour définir le cas des fonctions dans la relation de typage sémantique, on retombe sur une preuve élémentaire. De plus, la relation de typage sémantique définie à l'aide du jugement de typage se révèle stable par substitution de variables de types (proposition 1.5 ci-dessous), ce qui n'est pas le cas de la relation classique (définie sur les fonctions par continuité). Cette propriété est très utile dans la preuve de sûreté du typage. □

1.4.2 Généralisation sémantique

La proposition suivante est le lemme-clé pour montrer que la règle de typage du `let` est correcte.

Proposition 1.5 *Soient v une valeur et τ un type tels que $\models v : \tau$. Alors, pour toute substitution φ, on a $\models v : \varphi(\tau)$. En conséquence, pour tout ensemble de variables $\alpha_1 \dots \alpha_n$, on a $\models v : \forall\alpha_1 \dots \alpha_n.\, \tau$.*

Démonstration : la preuve procède par récurrence structurelle sur v (la même récurrence qui définit $\models v : \tau$).

• **Cas $v = cst$ et $\tau =$ `int` ou $\tau =$ `bool`.** Immédiat, puisqu'alors $\varphi(\tau) = \tau$.

• **Cas $v = (v_1, v_2)$ et $\tau = \tau_1 \times \tau_2$.** On applique l'hypothèse de récurrence à v_1 et v_2. Il vient $\models v_1 : \varphi(\tau_1)$ et $\models v_2 : \varphi(\tau_2)$. D'où le résultat.

• **Cas $v = (f, x, a, e)$ et $\tau = \tau_1 \to \tau_2$.** Soit E un environnement de typage tel que $\models e : E$ et $E \vdash (f \ \texttt{where}\ f(x) = a) : \tau_1 \to \tau_2$. Par la proposition 1.2, on a

$$\varphi(E) \vdash (f \ \texttt{where}\ f(x) = a) : \varphi(\tau_1 \to \tau_2).$$

Il ne reste plus qu'à établir $\models e : \varphi(E)$. Soit $y \in \mathrm{Dom}(e)$. On écrit $E(y)$ sous la forme $\forall\alpha_1 \dots \alpha_n.\, \tau_y$, avec les α_i choisis hors de portée de φ. On a donc $\varphi(E(y)) = \forall\alpha_1 \dots \alpha_n.\, \varphi(\tau_y)$. Il faut montrer $\models e(y) : \tau'$ pour toute instance τ' de $\varphi(E(y))$. Soit τ' une telle instance. On a donc $\tau' = \psi(\varphi(\tau_y))$ pour une certaine substitution ψ. On sait d'autre part que $\models e(y) : \tau_y$, par définition de $\models$ sur les schémas de types. On peut donc appliquer l'hypothèse de récurrence à la valeur $e(y)$, au type τ_y et à la substitution $\psi \circ \varphi$. Il vient $\models e(y) : \tau'$. Ceci pour toute instance τ' de $\varphi(E(y))$. D'où $\models e(y) : \varphi(E(y))$. Ceci pour tout $y \in \mathrm{Dom}(e)$. D'où $\models e : \varphi(E)$, et le résultat attendu. □

1.4.3 Preuve de sûreté

On peut maintenant énoncer plus précisément et prouver la sûreté du typage :

Proposition 1.6 (Sûreté forte) *Soient a une expression, τ un type, E un environnement de typage, et e un environnement d'évaluation tels que $E \vdash a : \tau$ et $\models e : E$. S'il existe une réponse r telle que $e \vdash a \Rightarrow r$, alors $r \neq$ `err` ; au contraire, r est égale à une valeur v, qui de plus vérifie $\models v : \tau$.*

Démonstration : la preuve procède par récurrence sur la taille de la dérivation d'évaluation. On raisonne par cas sur a, et donc sur la dernière règle utilisée dans la dérivation de typage.

• **Cas d'une constante.**

$$\frac{\tau \leq \texttt{TypCst}(cst)}{E \vdash cst : \tau}$$

La seule évaluation possible est $e \vdash cst \Rightarrow cst$. Reste à vérifier que $\models cst : \texttt{TypCst}(cst)$ pour toute constante cst. C'est vrai avec l'ensemble de constantes et le typage `TypCst` donnés en exemple plus haut.

• **Cas d'une variable.**

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

On sait par le typage que x appartient au domaine de E, qui est aussi le domaine de e par hypothèse $\models e : E$, donc la seule évaluation possible est $e \vdash x \Rightarrow e(x)$. Par hypothèse sur e et E, on a $\models e(x) : E(x)$. D'où $\models e(x) : \tau$ par définition de $\models$ sur les schémas de types.

• **Cas d'une fonction.**

$$\frac{E + f \mapsto (\tau_1 \to \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2}{E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \to \tau_2}$$

La seule évaluation possible est $e \vdash (f \ \texttt{where} \ f(x) = a) \Rightarrow (f, x, a, e)$. On a bien $\models (f, x, a, e) : \tau_1 \to \tau_2$ par définition de $\models$, en prenant E pour l'environnement de typage requis.

• **Cas d'une application.**

$$\frac{E \vdash a_1 : \tau_2 \to \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1}$$

On a trois possibilités d'évaluation. La première conclut $r = \texttt{err}$ parce que $e \vdash a_1 \Rightarrow r_1$ et r_1 n'est pas une fermeture ; mais elle est exclue par l'hypothèse de récurrence appliquée à a_1, qui nous dit que $\models r_1 : \tau_2 \to \tau_1$, et donc a fortiori r_1 est une fermeture. La deuxième possibilité d'évaluation conclut $r = \texttt{err}$ parce que $e \vdash a_2 \Rightarrow \texttt{err}$; elle est de même exclue par l'hypothèse de récurrence appliquée à a_2. On est donc dans le troisième cas d'évaluation :

$$\frac{e \vdash a_1 \Rightarrow (f, x, a_0, e_0) \qquad e \vdash a_2 \Rightarrow v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \Rightarrow r}{e \vdash a_1(a_2) \Rightarrow r}$$

On sait par hypothèse de récurrence que $\models (f, x, a_0, e_0) : \tau_2 \to \tau_1$, et que $\models v_2 : \tau_2$. Donc il existe E_0 tel que $\models e_0 : E_0$ et $E_0 \vdash (f \ \texttt{where} \ f(x) = a_0) : \tau_2 \to \tau_1$. Une seule règle de typage permet de dériver ce résultat ; est donc vraie sa prémisse :

$$E_0 + f \mapsto (\tau_2 \to \tau_1) + x \mapsto \tau_2 \vdash a_0 : \tau_1.$$

On considère alors les environnements

$$e_1 = e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \quad \text{et} \quad E_1 = E_0 + f \mapsto (\tau_2 \to \tau_1) + x \mapsto \tau_2.$$

On a $\models e_1 : E_1$ et $E_1 \vdash a_0 : \tau_1$. Appliquant une troisième fois l'hypothèse de récurrence, il vient $r \neq \texttt{err}$ et $\models r : \tau_1$, ce qui est le résultat attendu.

• **Cas du `let`.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

On a deux possibilités d'évaluation. La première correspond au cas $e \vdash a_1 \Rightarrow \texttt{err}$. Elle est exclue par l'hypothèse de récurrence appliquée à a_1. L'évaluation est donc de la forme :

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e + x \mapsto v_1 \vdash a_2 \Rightarrow r}{e \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 \Rightarrow r}$$

Par l'hypothèse de récurrence appliquée à a_1, il vient $\models v_1 : \tau_1$. Par la proposition 1.5, on a $\models v_1 : \texttt{Gen}(\tau_1, E)$. Notant

$$e_1 = e + x \mapsto v_1 \quad \text{et} \quad E_1 = E + x \mapsto \texttt{Gen}(\tau_1, E),$$

on a donc $\models e_1 : E_1$. On applique l'hypothèse de récurrence à a_2 considérée dans les environnements e_1 et E_1. Il vient le résultat attendu : $r \neq \texttt{err}$ et $\models r : \tau_2$.

• **Cas de la paire.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2}$$

Même raisonnement que pour l'application, en plus simple.

• **Cas des primitives.**

$$\frac{\tau_1 \to \tau_2 \leq \texttt{TypOp}(op) \qquad E \vdash a : \tau_1}{E \vdash op(a) : \tau_2}$$

Par hypothèse de récurrence, on a $e \vdash a \Rightarrow v_1$ avec $\models v_1 : \tau_1$. Le reste de la preuve dépend bien entendu des types et des comportements qu'on a donnés aux opérateurs. On se convainc aisément du résultat dans le cas de l'addition entière ou de la conditionnelle, les deux exemples donnés plus haut. □

La propriété de sûreté faible (proposition 1.4) s'ensuit immédiatement de la propriété de sûreté forte (proposition 1.6).

1.5 Inférence de types

Nous avons vu que si une expression close a est du type τ, elle est aussi du type τ' pour tout τ' moins général que τ (c'est-à-dire, $\tau' = \varphi(\tau)$ pour une substitution φ). Nous allons maintenant montrer que l'ensemble de tous les types de a est engendré de cette manière : il existe un type pour a, appelé le TYPE PRINCIPAL de a, qui est plus général que tous les types possibles pour a. Le calcul de ce type s'appelle l'INFÉRENCE DE TYPES. L'existence d'un algorithme d'inférence de types est la condition nécessaire pour qu'un compilateur ML puisse deviner les types des objets à partir de leurs utilisations, sans que le programmeur ait à fournir les types dans le programme source.

Commençons par quelques rappels sur la notion d'unificateur principal. On dit qu'une substitution φ est un UNIFICATEUR des deux types τ_1 et τ_2 si $\varphi(\tau_1) = \varphi(\tau_2)$. Deux types sont UNIFIABLES s'il existe un unificateur de ces deux types. Intuitivement, deux types sont unifiables si on peut les identifier en modifiant certaines de leurs variables de types. Un unificateur de deux types représente les modifications qu'il faut apporter à ces deux types pour les rendre égaux. On dit qu'un unificateur φ de τ_1 et τ_2 est PRINCIPAL si tout autre unificateur ψ de τ_1 et τ_2 se décompose en $\theta \circ \varphi$ pour une certaine substitution θ. L'unificateur principal de deux types, lorsqu'il existe, représente les modifications minimales qu'il faut apporter aux deux types pour les identifier : tout autre unificateur doit effectuer au moins ces modifications-là, plus d'autres.

Proposition 1.7 *Si deux types τ_1 et τ_2 sont unifiables, alors ils admettent un unificateur principal, unique à un renommage près, qu'on note* $\texttt{mgu}(\tau_1, \tau_2)$.

Démonstration : l'ensemble des types forme une algèbre libre de termes. L'unificateur principal se calcule par l'algorithme de Robinson [84], ou l'une de ses variantes. □

Parmi les unificateurs principaux de deux types τ_1 et τ_2, il en existe qui "n'introduisent pas de nouvelles variables", c'est-à-dire tels que toute variable non libre dans τ_1 ni dans τ_2 est hors de portée de l'unificateur. (C'est le cas pour l'unificateur construit par l'algorithme de Robinson.) Par la suite, on suppose que $\texttt{mgu}(\tau_1, \tau_2)$ est choisi de manière à vérifier cette propriété.

On donne maintenant une variante de l'algorithme de Damas et Milner [22], qui calcule le type principal d'une expression s'il existe. L'algorithme prend en entrée une expression a, un environnement de typage E et un ensemble de "nouvelles" variables V ; il retourne un type τ (le type le plus général pour a), une substitution φ (représentant les instanciations qu'on a dû effectuer dans E), et un sous-ensemble V' de V (les "nouvelles" variables qu'on n'a pas utilisées).

On note $\texttt{Inst}(\sigma, V)$ une instance triviale du schéma σ. C'est-à-dire, notant $\sigma = \forall \alpha_1 \dots \alpha_n . \tau$, on choisit n variables distinctes $\beta_1 \dots \beta_n$ dans V et on prend

$$\texttt{Inst}(\sigma, V) = ([\alpha_1 \mapsto \beta_1, \dots, \alpha_n \mapsto \beta_n](\tau),\ V \setminus \{\beta_1 \dots \beta_n\}).$$

Clairement, $\texttt{Inst}(\sigma, V)$ est défini à un renommage près des variables de V en variables de V.

Algorithme 1.1 $\texttt{Infer}(E, a, V)$ est le triplet (τ, φ, V') défini par :

Si a est x et $x \in \mathrm{Dom}(E)$:
$(\tau, V') = \texttt{Inst}(E(x), V)$ et $\varphi = [\,]$

Si a est cst :
$(\tau, V') = \texttt{Inst}(\texttt{TypCst}(cst), V)$ et $\varphi = [\,]$
Si a est $(f \ \texttt{where} \ f(x) = a_1)$:
soient α et β deux variables prises dans V
soit $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1,\ E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha,\ V \setminus \{\alpha, \beta\})$
soit $\mu = \texttt{mgu}(\varphi_1(\beta), \tau_1)$
alors $\tau = \mu(\varphi_1(\alpha \to \beta))$ et $\varphi = \mu \circ \varphi_1$ et $V' = V_1$
Si a est $a_1(a_2)$:
soit $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
soit $(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), V_1)$
soit α une variable de V_2
soit $\mu = \texttt{mgu}(\varphi_2(\tau_1), \tau_2 \to \alpha)$
alors $\tau = \mu(\alpha)$ et $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ et $V' = V_2 \setminus \{\alpha\}$
Si a est $\texttt{let} \ x = a_1 \ \texttt{in} \ a_2$:
soit $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
soit $(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2,\ \varphi_1(E) + x \mapsto \texttt{Gen}(\tau_1, \varphi_1(E)),\ V_1)$
alors $\tau = \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V = V_2$
Si a est (a_1, a_2) :
soit $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
soit $(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), V_1)$
alors $\tau = \tau_1 \times \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V' = V_2$
Si a est $op(a_1)$:
soit $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
soit $(\tau_2, V_2) = \texttt{Inst}(\texttt{TypOp}(op), V_1)$
soit α une variable de V_2
soit $\mu = \texttt{mgu}(\tau_1 \to \alpha, \tau_2)$
alors $\tau = \mu(\alpha)$ et $\varphi = \mu \circ \varphi_1$ et $V' = V_2 \setminus \{\alpha\}$

On convient que $\texttt{Infer}(a, E, V)$ n'est pas défini si, en cours de calcul, aucun cas ne s'applique ; en particulier, si on tente d'unifier deux types non unifiables.

Proposition 1.8 (Correction de l'algorithme d'inférence) *Soient a une expression, E un environnement de typage et V un ensemble de variables de types. Si $(\tau, \varphi, V') = \texttt{Infer}(a, E, V)$ est défini, alors on peut dériver $\varphi(E) \vdash a : \tau$.*

Démonstration : la preuve est par récurrence structurelle sur a, et utilise de manière essentielle la stabilité du typage par substitution (proposition 1.2). On donne un cas de base, et deux cas qui utilisent l'hypothèse de récurrence ; les autres cas sont similaires. On reprend les notations de l'algorithme.

• **Cas $a = x$.** On a $(\tau, V') = \texttt{Inst}(E(x), V)$ et $\varphi = [\,]$. Par définition de `Inst`, on a $\tau \leq E(x)$. On peut donc bien dériver $E \vdash x : \tau$.

• **Cas $a = (f \ \texttt{where} \ f(x) = a_1)$.** Appliquant l'hypothèse de récurrence à l'appel récursif de `Infer`, on obtient une preuve de

$$\varphi_1(E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha) \vdash a_1 : \tau_1.$$

Par la proposition 1.2, on en déduit une preuve de :

$$\varphi(E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha) \vdash a_1 : \mu(\tau_1).$$

La substitution μ étant par construction un unificateur de $\varphi_1(\beta)$ et de τ_1, on a $\varphi(\beta) = \psi(\tau_1)$. On a donc prouvé :

$$\varphi(E) + f \mapsto (\varphi(\alpha) \to \varphi(\beta)) + x \mapsto \varphi(\alpha) \vdash a_1 : \varphi(\beta).$$

Appliquant la règle de typage des fonctions, on déduit :

$$\varphi(E) \vdash (f \ \texttt{where} \ f(x) = a_1) : \varphi(\alpha) \to \varphi(\beta).$$

C'est le résultat attendu, puisque $\tau = \varphi(\alpha \to \beta)$.

• **Cas** $a = \texttt{let} \ x = a_1 \ \texttt{in} \ a_2$**.** On applique l'hypothèse de récurrence aux deux appels récursifs de `Infer`. Il vient des preuves de

$$\varphi_1(E) \vdash a_1 : \tau_1 \qquad \varphi_2(\varphi_1(E) + x \mapsto \texttt{Gen}(\tau_1, \varphi_1(E))) \vdash a_2 : \tau_2.$$

Si nécessaire, on renomme dans la dérivation de gauche les variables généralisées pour qu'elles soient hors de portée de φ_2. On montre alors

$$\texttt{Gen}(\varphi_2(\tau_1), \varphi_2(\varphi_1(E))) = \varphi_2(\texttt{Gen}(\tau_1, \varphi_1(E)))$$

de la même manière que dans la preuve de la proposition 1.2, cas `let`. Notant $\varphi = \varphi_2 \circ \varphi_1$, on a donc des preuves de :

$$\varphi(E) \vdash a_1 : \varphi_2(\tau_1) \qquad \varphi(E) + x \mapsto \texttt{Gen}(\varphi_2(\tau_1), \varphi(E)) \vdash a_2 : \tau_2.$$

On conclut, par la règle de typage du `let`,

$$\varphi(E) \vdash \texttt{let} \ x = a_1 \ \texttt{in} \ a_2 : \tau_2.$$

C'est le résultat annoncé. □

Proposition 1.9 (Complétude de l'algorithme d'inférence) *Soient a une expression, E un environnement de typage, et V un ensemble infini de variables telles que $V \cap \mathcal{L}(E) = \emptyset$. S'il existe un type τ' et une substitution φ' tels que $\varphi'(E) \vdash a : \tau'$, alors $(\tau, \varphi, V') = \texttt{Infer}(a, E, V)$ est bien défini, et il existe une substitution ψ telle que*

$$\tau' = \psi(\tau) \quad \textit{et} \quad \varphi' = \psi \circ \varphi \ \textit{hors de} \ V.$$

La notation "$\varphi' = \psi \circ \varphi$ HORS DE V" signifie que $\varphi'(\alpha)$ est égal à $\psi(\varphi(\alpha))$ pour toute variable α n'appartenant pas à V. La condition $\varphi' = \psi \circ \varphi$ hors de V signifie que les substitutions φ' et $\psi \circ \varphi$ se comportent de la même manière sur le problème de typage initial. Les variables de V sont des variables qui n'apparaissent pas dans le problème initial, mais peuvent être introduites comme intermédiaires de calcul au cours de l'inférence ; on n'a donc pas à en tenir compte pour comparer φ' (la solution proposée au problème de typage) et φ (la solution inférée).

Démonstration : on commence par remarquer que, avec les hypothèses de la proposition, si $(\tau, \varphi, V') = \texttt{Infer}(a, E, V)$ est défini, alors $V' \subseteq V$, et les variables de V' ne sont pas libres dans τ et sont hors de portée de φ. En conséquence, $V' \cap \mathcal{L}(\varphi(E)) = \emptyset$.

La preuve de la proposition 1.9 est par récurrence structurelle sur a. On donne un cas de base et trois cas de récurrence ; les autres cas sont similaires.

• **Cas** $a = x$**.** Puisque $\varphi(E) \vdash x : \tau$, on a $x \in \mathrm{Dom}(\varphi(E))$ et $\tau \leq \varphi(E)(x)$. Ceci entraîne $x \in \mathrm{Dom}(E)$. Donc $\texttt{Infer}(E, x)$ est défini et renvoie

$$\tau = [\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n](\tau_x) \quad \text{et} \quad \varphi = [\,] \quad \text{et} \quad V' = V \setminus \{\beta_1 \ldots \beta_n\}$$

pour certaines variables $\beta_1 \ldots \beta_n \in V$. Écrivons $E(x) = \forall \alpha_1 \ldots \alpha_n . \tau_x$, avec les α_i choisies dans V' et hors de portée de φ'. On a donc $\varphi'(E(x)) = \forall \alpha_1 \ldots \alpha_n . \varphi'(\tau_x)$. On note ρ la substitution sur les α_i telle que $\tau' = \rho(\varphi'(\tau_x))$. On prend

$$\psi = \rho \circ \varphi' \circ [\beta_1 \mapsto \alpha_1, \ldots, \beta_n \mapsto \alpha_n].$$

On a $\psi(\tau) = \rho(\varphi'(\tau_x)) = \tau'$. D'autre part, toute variable $\alpha \notin V$ n'est ni une des α_i, ni une des β_i, d'où $\psi(\alpha) = \rho(\varphi'(\alpha)) = \varphi'(\alpha)$. C'est le résultat annoncé, puisque $\varphi = [\,]$ ici.

• **Cas** $a = (f \ \texttt{where} \ f(x) = a_1)$**.** La dérivation initiale se termine par

$$\frac{\varphi'(E) + f \mapsto (\tau'_2 \to \tau'_1) + x \mapsto \tau'_2 \vdash a_1 : \tau'_1}{\varphi'(E) \vdash (f \ \texttt{where} \ f(x) = a) : \tau'_2 \to \tau'_1}$$

Prenons α et β dans V, comme dans l'algorithme. On définit l'environnement E_1 et la substitution φ'_1 par

$$E_1 = E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha \quad \text{et} \quad \varphi'_1 = \varphi' + \alpha \mapsto \tau'_2 + \beta \mapsto \tau'_1.$$

On a $\varphi'_1(E_1) = \varphi'(E) + f \mapsto (\tau'_2 \to \tau'_1) + x \mapsto \tau'_2$. On applique l'hypothèse de récurrence à a_1, E_1, $V \setminus \{\alpha, \beta\}$, φ'_1 et τ'_2. Il vient

$$(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E_1, V \setminus \{\alpha, \beta\}) \quad \text{et} \quad \tau'_1 = \psi_1(\tau_1) \quad \text{et} \quad \varphi'_1 = \psi_1 \circ \varphi_1 \text{ hors de } V \setminus \{\alpha, \beta\}.$$

En particulier, on a $\psi_1(\varphi_1(\beta)) = \varphi'_1(\beta) = \tau'_1$, donc ψ_1 est un unificateur de $\varphi_1(\beta)$ et de τ_1. L'unificateur principal de ces deux types existe donc — appelons-le μ —, et $\texttt{Infer}(E, a, v)$ est bien défini. Soit ψ une substitution telle que $\psi_1 = \psi \circ \mu$. Montrons que cette substitution ψ convient. On a :

$$\begin{array}{rcll} \psi(\tau) & = & \psi(\mu(\varphi_1(\alpha \to \beta))) & \text{par définition de } \tau \text{ dans l'algorithme} \\ & = & \psi_1(\varphi_1(\alpha \to \beta)) & \text{par définition de } \psi \\ & = & \varphi'_1(\alpha \to \beta) & \text{parce que } \alpha \text{ et } \beta \text{ sont hors de } V \setminus \{\alpha, \beta\} \\ & = & \tau'_2 \to \tau'_1 & \text{par construction de } \varphi'_1 \end{array}$$

D'autre part, pour toute variable γ hors de V,

$$\begin{array}{rcll} \psi(\varphi(\gamma)) & = & \psi(\mu(\varphi_1(\gamma))) & \text{par définition de } \varphi \text{ dans l'algorithme} \\ & = & \psi_1(\varphi_1(\gamma)) & \text{par définition de } \psi_1 \\ & = & \varphi'_1(\gamma) & \text{puisque } \gamma \notin V \\ & = & \varphi_1(\gamma) & \text{puisque } \gamma \notin V \text{ implique } \gamma \neq \alpha \text{ et } \gamma \neq \beta \end{array}$$

D'où le résultat annoncé.

• **Cas** $a = a_1(a_2)$**.** La dérivation initiale est de la forme

$$\frac{\varphi'(E) \vdash a_1 : \tau'' \to \tau' \qquad \varphi'(E) \vdash a_2 : \tau''}{\varphi'(E) \vdash a_1(a_2) : \tau'}$$

On applique l'hypothèse de récurrence à a_1, E, V, $\tau' \to \tau''$ et φ'. Il vient

$$(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V) \quad \text{et} \quad \tau'' \to \tau' = \psi_1(\tau_1) \quad \text{et} \quad \varphi' = \psi_1 \circ \varphi_1 \text{ hors de } V.$$

En particulier, $\varphi'(E) = \psi_1(\varphi_1(E))$. On applique l'hypothèse de récurrence à a_2, $\varphi_1(E)$, V_1, τ et ψ_1. On a bien $\mathcal{L}(\varphi_1(E)) \cap V_1 = \emptyset$ par la remarque du début de la preuve. Il vient :

$$(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), V_1) \quad \text{et} \quad \tau'' = \psi_2(\tau_2) \quad \text{et} \quad \psi_1 = \psi_2 \circ \varphi_2 \text{ hors de } V_1.$$

On a $\mathcal{L}(\tau_1) \cap V_1 = \emptyset$, d'où $\psi_1(\tau_1) = \psi_2(\varphi_2(\tau_1))$. Posons $\psi_3 = \psi_2 + \alpha \mapsto \tau'$. (La variable α, choisie dans V_2, est hors de portée de ψ_2, et donc ψ_3 prolonge ψ_2.) On a :

$$\begin{array}{lcccl} \psi_3(\varphi_2(\tau_1)) & = & \psi_2(\varphi_2(\tau_1)) & = & \psi_1(\tau_1) \quad = \quad \tau'' \to \tau' \\ \psi_3(\tau_2 \to \alpha) & = & \psi_2(\tau_2) \to \tau'' & = & \tau'' \to \tau' \end{array}$$

La substitution ψ_3 est donc un unificateur de $\varphi_2(\tau_1)$ et $\tau_2 \to \alpha$. L'unificateur principal de ces deux types, μ, existe donc, et $\texttt{Infer}(a_1(a_2), E, V)$ est bien défini. De plus, on a $\psi_3 = \psi_4 \circ \mu$ pour une certaine substitution ψ_4. On montre maintenant que $\psi = \psi_4$ convient. Avec les notations de l'algorithme, on a bien

$$\psi(\tau) = \psi_4(\mu(\alpha)))) = \psi_3(\alpha) = \tau',$$

d'une part, et d'autre part pour tout $\beta \notin V$ (et donc a fortiori $\beta \notin V_1$, $\beta \notin V_2$, $\beta \neq \alpha$) :

$$\begin{array}{lcll} \psi(\varphi(\beta)) & = & \psi_4(\mu(\varphi_2(\varphi_1(\beta)))) & \text{par définition de } \varphi \\ & = & \psi_3(\varphi_2(\varphi_1(\beta))) & \text{par définition de } \psi_4 \\ & = & \psi_2(\varphi_2(\varphi_1(\beta))) & \text{parce que } \beta \neq \alpha \text{ et } \alpha \text{ hors de portée de } \varphi_1 \text{ et de } \varphi_2 \\ & = & \psi_1(\varphi_1(\beta)) & \text{parce que } \varphi_1(\beta) \notin V_1 \\ & = & \varphi'(\beta) & \text{parce que } \beta \notin V. \end{array}$$

C'est le résultat annoncé.

• **Cas** $a = (\texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2)$**.** La dérivation initiale se termine par

$$\frac{\varphi'(E) \vdash a_1 : \tau' \qquad \varphi'(E) + x \mapsto \texttt{Gen}(\tau', \varphi'(E)) \vdash a_2 : \tau''}{\varphi'(E) \vdash \texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2 : \tau''}$$

On applique l'hypothèse de récurrence à a_1, E, V, τ' et φ'. Il vient

$$(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V) \quad \text{et} \quad \tau' = \psi_1(\tau_1) \quad \text{et} \quad \varphi' = \psi_1 \circ \varphi_1 \text{ hors de } V.$$

En particulier, $\varphi'(E) = \psi_1(\varphi_1(E))$. On vérifie facilement que $\psi_1(\mathtt{Gen}(\tau_1, \varphi_1(E)))$ est plus général que $\mathtt{Gen}(\psi_1(\tau_1), \psi_1(\varphi_1(E)))$, c'est-à-dire que $\mathtt{Gen}(\tau', \varphi'(E))$. Puisqu'on peut prouver

$$\varphi'(E) + x \mapsto \mathtt{Gen}(\tau', \varphi'(E)) \vdash a_2 : \tau'',$$

la proposition 1.3 dit qu'on peut a fortiori prouver

$$\varphi'(E) + x \mapsto \psi_1(\mathtt{Gen}(\tau_1, \varphi_1(E))) \vdash a_2 : \tau'',$$

c'est-à-dire

$$\psi_1(\varphi_1(E) + x \mapsto \mathtt{Gen}(\tau_1, \varphi_1(E))) \vdash a_2 : \tau''.$$

On applique l'hypothèse de récurrence à a_2, dans l'environnement $\varphi_1(E) + x \mapsto \mathtt{Gen}(\tau_1, \varphi_1(E))$, avec les variables V_1, le type τ'' et la substitution ψ_1. Il vient

$$(\tau_2, \varphi_2, V_2) = \mathtt{Infer}(a_2,\ \varphi_1(E) + x \mapsto \mathtt{Gen}(\tau_1, \varphi_1(E)),\ V_1)$$

et $\tau'' = \psi_2(\tau_2)$ et $\psi_1 = \psi_2 \circ \varphi_2$ hors de V_1. L'algorithme prend $\tau = \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V' = V_2$. Montrons que $\psi = \psi_2$ convient. On a bien $\psi(\tau) = \tau''$. Et si $\alpha \notin V$, a fortiori $\alpha \notin V_1$, et donc :

$$\begin{array}{rcll} \psi(\varphi(\alpha)) & = & \psi_2(\varphi_2(\varphi_1(\alpha)))) & \text{par définition de } \varphi \\ & = & \psi_1(\varphi_1(\alpha)) & \text{parce que } \varphi_1(\alpha) \notin V_1, \text{ puisque } \alpha \text{ hors de portée de } \varphi_1 \\ & = & \varphi'(\alpha) & \text{parce que } \alpha \notin V. \end{array}$$

D'où $\varphi' = \psi \circ \varphi$ hors de V, comme annoncé. □

Chapitre 2

Trois extensions en quête de typage

Dans ce chapitre, on enrichit le langage applicatif simplifié du chapitre précédent par trois traits importants des "vrais" langages algorithmiques. Premièrement, la possibilité de modifier les structures de données en place. Deuxièmement, la possibilité d'exécuter des morceaux de programmes en parallèle, tout en les faisant communiquer entre eux. Troisièmement, un certain nombre de structures de contrôle non-locales (les exceptions, les coroutines, voire — quelle horreur ! — certaines formes de `goto`).

On fait apparaître ces traits en introduisant trois nouvelles sortes d'objets "de première classe", c'est-à-dire des objets qu'on peut manipuler comme n'importe quelle autre valeur du langage : les références ou cellules d'indirection, pour la modification physique ; les canaux de communication, pour les processus communicants ; et les continuations, pour les structures de contrôle non-locales. On va ajouter au langage purement applicatif des primitives pour créer et pour manipuler ces objets, et donner la sémantique de ces primitives.

Du point de vue du typage, ces trois extensions présentent des similitudes frappantes. Il y a une manière naturelle de les typer, par l'introduction de nouveaux constructeurs de types unaires. Les règles de typage qui en découlent sont simples et intuitives. Les systèmes de types obtenus sont sûrs en l'absence de polymorphisme. Cependant, on constate qu'ils ne sont pas sûrs en présence de polymorphisme. On se contente, dans ce chapitre, de donner des contre-exemples, réservant aux chapitres suivants une discussion plus approfondie du phénomène, et l'étude de remèdes.

2.1 Les références

2.1.1 Présentation

Les références sont des cellules d'indirection modifiables en place. Une référence a un contenu courant (une valeur quelconque du langage), qui peut être lu et écrit à tout instant. Une référence est elle-même une valeur : on peut la renvoyer en résultat d'une fonction, ou la mettre dans une structure

de données. Les références s'introduisent par le biais de trois nouvelles opérations primitives :

`Op`	::=	`ref`	création d'une nouvelle référence
	|	!	lecture du contenu courant
	|	:=	écriture du contenu courant

L'expression `ref`(a) renvoie une nouvelle référence, initialisée à la valeur de a. L'expression !(a), qu'on écrit plus volontiers !a, évalue a en une référence et renvoie son contenu courant. Quant à l'expression $:= (a_1, a_2)$, qu'on préfère écrire $a_1 := a_2$, elle évalue a_1 en une référence et remplace physiquement son contenu par la valeur de a_2. L'expression $a_1 := a_2$ elle-même n'a pas de valeur qui lui est naturellement associée : elle opère essentiellement par effet de bord. On convient qu'elle renvoie une nouvelle constante, notée `()` (lire : "vide").

`Cst`	::=	`...`	
	|	`()`	la valeur vide

Exemple. Les références plus le `let` donnent les variables modifiables (les variables vraiment variables). Voici par exemple comment faire une boucle pour `!i` variant de 0 à `n` $- 1$:

```
let i = ref(0) in
  while !i < n do ...; i := 1 + !i done
```

Ici et par la suite, on note $a_1; a_2$ la construction qui évalue d'abord a_1, puis a_2, et renvoie la valeur de a_2. On peut voir cette construction comme une abréviation de `let` $z = a_1$ `in` a_2, où z est un nouvel identificateur, non libre dans a_2. De même, la boucle `while` a_1 `do` a_2 `done` est une abréviation pour

$$(f \texttt{ where } f(z) = \texttt{if } a_1 \texttt{ then } a_2; f(z) \texttt{ else } ())(())$$

où f et z sont de nouveaux identificateurs. □

Exemple. Les références plus les structures de données donnent les structures de données modifiables en place. Par exemple, pour représenter une algèbre de termes, on peut représenter les variables par des références vers une constante particulière. Pour substituer une variable par un terme, il suffit de faire pointer la référence correspondante vers le terme [11]. La substitution d'une variable s'effectue donc en temps constant ; au contraire, dans le cas d'une représentation des termes sans références, il aurait fallu recopier entièrement le terme dans lequel on substitue la variable, ce qui est moins efficace. De plus, si la variable substituée est commune à plusieurs termes, la substitution prend effet dans tous les termes qui contiennent la variable avec la solution à base de références. Dans la solution sans référence, il faut, pour obtenir le même effet, substituer explicitement tous les termes contenant la variable substituée, obligeant le programmeur à garder trace de tous ces termes. □

Exemple. Les références plus les fonctions donnent les fonctions avec état — une notion proche de celle d'objet dans les langages dits *object-oriented*. Voici par exemple un générateur de nombres pseudo-aléatoires : une fonction qui renvoie, à chaque appel, l'élément suivant d'une suite difficile à prévoir. On fournit aussi une fonction de réinitialisation du générateur.

```
let (random, set_random) =
  let seed = ref 0 in
    (λx. seed := (!seed × 25173 + 13849) mod 1000; !seed),
    (λnewseed. seed := newseed)
in ...
```

(On s'est permis un `let` avec déstructuration d'une paire, dont la signification est évidente.) Cet exemple se programme difficilement dans un langage purement applicatif. La fonction `random` doit alors prendre la variable d'état `seed` en argument, et renvoyer sa nouvelle valeur en résultat. Le programme utilisant `random` a la charge de propager correctement cette valeur, c'est-à-dire d'utiliser la valeur retournée par le précédent appel de `random` comme argument du prochain appel. Ceci encombre désagréablement le programme utilisateur. De plus, il est facile de se tromper et de passer à `random` toujours la même valeur de `seed`. La solution avec des références est d'un bien meilleur style, plus modulaire et plus sûr. □

2.1.2 Sémantique

On va maintenant donner une sémantique relationnelle au langage avec références. Les références nécessitent d'introduire dans la sémantique une notion d'état global (les contenus courants de toutes les références) qui évolue au cours de l'évaluation. En termes plus imagés, on considère les références comme des adresses de la cellule dans une mémoire centrale. La sémantique doit non seulement attribuer une valeur à chaque expression, mais aussi décrire comment l'état de la mémoire est modifié par l'évaluation de l'expression. On se donne donc un ensemble infini d'ADRESSES MÉMOIRE ℓ (*locations*), et on représente les ÉTATS MÉMOIRE s (*stores*) comme des applications finies des adresses dans les valeurs. Voici donc les objets sémantiques utilisés :

Résultat :	r	::=	v/s	résultat normal
		|	`err`	résultat d'erreur
Valeurs :	v	::=	cst	valeur de base
		|	(v_1, v_2)	paire de valeurs
		|	(f, x, a, e)	valeur fonctionnelle
		|	ℓ	adresse mémoire
Environnements :	e	::=	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	
Etats mémoire :	s	::=	$[\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n]$	

Le prédicat d'évaluation est maintenant de la forme $e \vdash a/s_0 \Rightarrow r$, c'est-à-dire "dans l'environnement d'évaluation e, l'expression a considérée dans l'état mémoire initial s_0 s'évalue en la réponse r." Une réponse est soit `err`, soit un couple v/s_1, où v est la valeur résultante et s_1 l'état mémoire à la fin de l'évaluation.

Voici les règles définissant le nouveau prédicat d'évaluation. On donne sans commentaires les règles pour les constructions purement applicatives. Les constantes et les variables :

$$e \vdash cst/s \Rightarrow cst/s \qquad \frac{x \in \mathrm{Dom}(e)}{e \vdash x/s \Rightarrow e(x)/s} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x/s \Rightarrow \texttt{err}}$$

L'abstraction fonctionnelle :

$$e \vdash (f \texttt{ where } f(x) = a)/s \Rightarrow (f, x, a, e)/s$$

L'application :

$$\frac{e \vdash a_1/s_0 \Rightarrow (f, x, a_0, e_0)/s_1 \qquad e \vdash a_2/s_1 \Rightarrow v_2/s_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0/s_2 \Rightarrow r_0}{e \vdash a_1(a_2)/s_0 \Rightarrow r_0}$$

$$\frac{e \vdash a_1/s_0 \Rightarrow r_1 \qquad r_1 \text{ n'est pas de la forme } (f, x, a_0, e_0)/s_1}{e \vdash a_1(a_2)/s_0 \Rightarrow \texttt{err}}$$

$$\frac{e \vdash a_1/s_0 \Rightarrow (f, x, a_0, e_0)/s_1 \qquad e \vdash a_2/s_1 \Rightarrow \texttt{err}}{e \vdash a_1(a_2)/s_0 \Rightarrow \texttt{err}}$$

La construction `let` :

$$\frac{e \vdash a_1/s_0 \Rightarrow v_1/s_1 \qquad e + x \mapsto v_1 \vdash a_2/s_1 \Rightarrow r_2}{e \vdash (\texttt{let } x = a_1 \texttt{ in } a_2)/s_0 \Rightarrow r_2} \qquad \frac{e \vdash a_1/s_0 \Rightarrow \texttt{err}}{e \vdash (\texttt{let } x = a_1 \texttt{ in } a_2)/s_0 \Rightarrow \texttt{err}}$$

La paire :

$$\frac{e \vdash a_1/s_0 \Rightarrow v_1/s_1 \qquad e \vdash a_2/s_1 \Rightarrow v_2/s_2}{e \vdash (a_1, a_2)/s_0 \Rightarrow (v_1, v_2)/s_2}$$

$$\frac{e \vdash a_1/s_0 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2)/s_0 \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_1/s_0 \Rightarrow v_1/s_1 \qquad e \vdash a_2/s_1 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2)/s_0 \Rightarrow \texttt{err}}$$

On détaille maintenant un peu plus les règles pour les références.

$$\frac{e \vdash a/s_0 \Rightarrow v/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash \texttt{ref}(a)/s_0 \Rightarrow \ell/(s_1 + \ell \mapsto v)} \qquad \frac{e \vdash a/s_0 \Rightarrow \texttt{err}}{e \vdash \texttt{ref}(a)/s_0 \Rightarrow \texttt{err}}$$

La création d'une référence évalue d'abord la valeur d'initialisation v, puis choisit une adresse mémoire ℓ non encore occupée (c'est toujours possible, car on s'est donné un nombre infini d'adresses), et enrichit l'état mémoire de la liaison $\ell \mapsto v$. La partie valeur du résultat est l'adresse ℓ.

Remarque. Cette règle peut sembler détruire l'aspect déterministe de l'évaluation : on peut en effet choisir différentes adresses ℓ. Pourtant, tous ces choix se valent, en un certain sens : si $e \vdash a/s_0 \Rightarrow v/s_1$, alors la réponse v/s_1 est unique modulo un renommage des adresses occupées dans s_1 mais non dans s_0. □

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash !a/s_0 \Rightarrow s_1(\ell)/s_1}$$

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash\, !a/s_0 \Rightarrow \texttt{err}} \qquad \frac{e \vdash a/s_0 \Rightarrow r \qquad r \text{ n'est pas de la forme } \ell/s_1}{e \vdash\, !a/s_0 \Rightarrow \texttt{err}}$$

L'opérateur de lecture évalue son argument, puis consulte la valeur stockée à l'adresse mémoire obtenue. C'est une erreur si l'adresse tombe en-dehors de la mémoire occupée après évaluation de l'argument.

$$\frac{e \vdash a/s_0 \Rightarrow (\ell, v)/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash :=(a)/s_0 \Rightarrow \texttt{()}/(s_1 + \ell \mapsto v)}$$

$$\frac{e \vdash a/s_0 \Rightarrow (\ell, v)/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash :=(a)/s_0 \Rightarrow \texttt{err}} \qquad \frac{e \vdash a/s_0 \Rightarrow r \qquad r \text{ n'est pas de la forme } (\ell, v)/s_1}{e \vdash :=(a)/s_0 \Rightarrow \texttt{err}}$$

L'opérateur d'écriture évalue ses deux arguments, le premier devant être une adresse ℓ déjà occupée. Il modifie alors l'état mémoire au point ℓ, remplaçant l'ancienne valeur par la valeur nouvellement calculée.

2.1.3 Typage

Le typage naturel des références est le suivant : on introduit dans les expressions de types la construction τ `ref`, qui est le type des références contenant une valeur de type τ.

$$\begin{array}{rcll} \tau & ::= & \ldots & \\ & | & \tau\ \texttt{ref} & \text{type d'une référence} \end{array}$$

Dès lors, le résultat de la création d'une référence `ref`(a) est de type τ `ref`, pourvu que a soit de type τ. La lecture d'une référence $!a$ renvoie un objet de type τ, pourvu que a soit de type τ `ref`. Finalement, l'écriture d'une référence $a_1 := a_2$ est correcte si a_2 est bien du type attendu pour le contenu de a_1, c'est-à-dire si $a_1 : \tau$ `ref` et $a_2 : \tau$ pour un certain type τ. Tout ceci se résume en prenant :

$$\begin{array}{rcl} \texttt{TypOp(ref)} & = & \forall\alpha.\, \alpha \to \alpha\ \texttt{ref} \\ \texttt{TypOp(!)} & = & \forall\alpha.\, \alpha\ \texttt{ref} \to \alpha \\ \texttt{TypOp(:=)} & = & \forall\alpha.\, \alpha\ \texttt{ref} \times \alpha \to \texttt{unit} \end{array}$$

Dans le type de :=, on a utilisé un nouveau type de base, `unit`, qui est le type contenant la seule valeur `()` :

$$\begin{array}{rcl} \texttt{TypBas} & = & \{\texttt{unit}; \texttt{int}; \texttt{bool}; \ldots\} \\ \texttt{TypCst(())} & = & \texttt{unit} \end{array}$$

Les types proposés ci-dessus pour les primitives sur les références peuvent sembler tout à fait justes. On montre assez facilement qu'ils sont corrects en l'absence de polymorphisme. Cependant, ce typage se révèle incorrect en présence de polymorphisme : une référence munie d'un type polymorphe suffit à compromettre la sûreté de l'exécution.

Exemple. Le programme suivant est bien typé avec les types ci-dessus :

```
let r = ref(λx.x) in
    r := (λn.n + 1);
    if (!r)(true) then ... else ...
```

Le type de `r` dans le corps du `let` est en effet $\forall\alpha.\,(\alpha \to \alpha)$ `ref`. On peut donc employer `r` une première fois avec le type (`int` $\to$ `int`) `ref`, et y stocker la fonction successeur ; puis une deuxième fois avec le type (`bool` $\to$ `bool`) `ref`, et appliquer le contenu de `r` à `true`. À l'exécution, on se retrouve cependant en train d'additionner 1 et `true` — une erreur de type, s'il en est. □

Il semble clair qu'une référence, une fois créée, doit toujours être utilisée avec le même type ; autrement, on permet d'écrire dans la référence un objet d'un certain type, puis de le relire en prétendant qu'il a un autre type. Cette propriété de cohérence entre les utilisations d'une même référence n'est évidemment plus assurée dès qu'on se permet des références polymorphes (c'est-à-dire, auxquelles on a attribué un schéma de types non trivial). Il va donc falloir modifier le système de types pour interdire les références polymorphes ; c'est l'objet des deux chapitres suivants.

2.2 Les canaux de communication

2.2.1 Présentation

Jusqu'ici, on a considéré seulement des programmes qui vivent dans un monde clos : leur évaluation ne fait à aucun moment intervenir l'état du monde extérieur au programme. Les "vrais" programmes ne procèdent pas ainsi ; ils s'exécutent en interaction avec des entités extérieures : l'utilisateur, le système d'exploitation, d'autres programmes. On veut donc pouvoir décrire, dans le langage et dans sa formalisation, les interactions avec le monde extérieur. L'étape suivante est d'internaliser cette notion : on considère le programme non comme une expression monolithique qui s'évalue séquentiellement, mais comme plusieurs expressions qui s'évaluent parallèlement, en échangeant des informations entre elles. De nombreux algorithmes s'expriment alors plus simplement — en particulier, ceux qui requièrent un entrelacement compliqué de calculs. On s'attend aussi à une évaluation plus rapide, si l'on dispose de plusieurs processeurs entre lesquels répartir l'évaluation.

On va introduire ces notions dans le petit langage du chapitre 1 par le biais des canaux de communication. Un canal est un nouvel objet "de première classe", sur lequel on peut demander à envoyer une valeur, ou à recevoir une valeur. Une valeur se transmet effectivement lorsque deux processus demandent simultanément l'un à recevoir, l'autre à envoyer, sur le même canal. Les canaux permettent donc la synchronisation entre processus (mécanisme dit "du rendez-vous"), en plus de la communication. On fournit les primitives suivantes sur les canaux :

`Op`	::=	`newchan`	création d'un nouveau canal
	|	?	réception sur un canal
	|	!	émission sur un canal

L'usage est de noter $a?$ au lieu de $?(a)$ la réception depuis le canal a, et $a_1\,!\,a_2$ au lieu de $!(a_1, a_2)$ l'envoi de la valeur de a_2 sur le canal a_1. L'expression $a?$ s'évalue en la valeur reçue ; l'expression $a_1\,!\,a_2$, en `()`.

Pour communiquer par rendez-vous à travers un canal, encore faut-il pouvoir évaluer en parallèle deux expressions : une qui va émettre, l'autre qui va recevoir. On introduit donc deux constructions supplémentaires dans les expressions :

$$\begin{array}{rcll} a & ::= & \ldots & \\ & | & a_1 \parallel a_2 & \text{composition parallèle} \\ & | & a_1 \oplus a_2 & \text{choix non déterministe} \end{array}$$

L'expression $a_1 \parallel a_2$ évalue a_1 et a_2 en parallèle, et renvoie la paire des résultats. Elle permet à a_1 et a_2 de communiquer entre elles par rendez-vous. L'expression $a_1 \oplus a_2$ évalue ou bien a_1, ou bien a_2, le choix étant fait de manière non-déterministe. Cette construction sert à offrir plusieurs possibilités de communication à l'extérieur. La troisième manière de composer les évaluations, la séquence $(a_1; a_2)$, se définit facilement avec la construction `let`, comme on l'a vu plus haut.

Contexte. On a envisagé quantité de présentations très variées de la notion de processus communicants. Par ordre à peu près chronologique : les modèles à mémoire partagée plus sémaphores ou moniteurs ou verrous ; les réseaux de Petri ; les modèles à canaux (inspirés par les *streams* de Multics et les *pipes* de Unix) : les réseaux de processus de Kahn-MacQueen [43], le calcul CSP de Hoare [37], le calcul CCS de Milner [59] et ses nombreuses variantes ; plus récemment, les modèles à "données actives" [15, 6, 9, 45].

J'ai suivi d'aussi près que possible l'approche prise par Milner pour son *Calculus of Communicating Systems* [59]. La principale différence est que, ici, les canaux sont des valeurs de première classe, qui sont crées par la primitive `newchan`, et soumises aux règles de portée lexicale usuelles. Au contraire, en CCS, les canaux ne sont pas des valeurs ; ils sont associé de manière permanente à des noms globaux ; et pour limiter leur portée, il faut faire appel aux constructions de renommage et d'effacement, qui encodent une notion de portée dynamique.

Le calcul présenté ici est dit d'ordre supérieur, puisqu'on peut faire passer des valeurs de canaux à travers un canal. Il n'y a pas de consensus sur la manière d'étendre CCS à l'ordre supérieur. Thomsen [90] et Berthomieu [10] proposent d'avoir les processus comme valeurs, mais pas les canaux. Dans son pi-calcul [61], Milner propose d'avoir les noms de canaux comme valeurs, mais pas les canaux eux-mêmes — et débouche sur un calcul différent. Le principe "les canaux sont des valeurs" semble plus proche des mécanismes fournis par les systèmes d'exploitation — les *pipes* de Unix, en particulier.

Deux propositions récentes d'extension de ML par des primitives de parallélisme prennent les canaux comme valeur de première classe, comme je l'ai fait ici, mais fournissent en plus un type concret des événements de communication, qu'on peut combiner pour définir de nouveaux mécanismes de synchronisation. Il s'agit de la bibliothèque Concurrent ML de Reppy [80], reprise et simplifiée ensuite par Berry, Milner et Turner [8]. □

Exemple. La fonction `stamp` ci-dessous envoie la suite des entiers sur le premier des deux canaux passé en argument, et repart de zéro lorsqu'elle reçoit quelque chose sur le second canal. Mise en parallèle avec d'autres processus, elle peut servir de générateur de marques (*stamps*) uniques.

```
let stamp = λoutput. λreset.
```

```
    (f where f(n) = (output ! n; f(n + 1)) ⊕ (reset?; f(0)))(0) in
let st = newchan(()) in
let reset = newchan(()) in
    stamp(st)(reset) ‖ (reset ! (); ( ... st ? ... st ? ... ) ‖ ( ... st ? ... ))
```

□

Exemple. Le programme suivant, repris de [43], envoie la suite des nombres premiers sur le canal donné en argument.

```
let sieve = λprimes.
  let integers = newchan(()) in
    ((enumerate where enumerate(n) = integers ! n; enumerate(n + 1))(2)) ‖
    ((filter where filter(input) =
        let n = input ? in
          primes ! n;
          let output = newchan(()) in
            filter(output) ‖
            while true do
              let m = input ? in if m mod n = 0 then () else output ! m
            done)
     (integers))
in ...
```

On y reconnaît aisément l'algorithme d'Eratosthène. Un tel programme, ne bornant pas a priori les nombres premiers produits, est considérablement plus difficile à écrire dans un langage purement séquentiel (du moins en évaluation stricte). □

2.2.2 Sémantique

On donne maintenant une sémantique relationnelle au langage avec canaux. Le prédicat d'évaluation doit prendre en compte l'état courant des processus qui s'évaluent en parallèle avec l'expression considérée, afin de savoir quelles sont les possibilités courantes de communications. Pour ce faire, on ajoute un argument au prédicat d'évaluation, qui devient $e \vdash a \overset{w}{\Longrightarrow} r$. Ici, w est une suite finie d'ÉVÉNEMENTS ; cette suite représente l'ensemble des communications qui doivent avoir lieu pour arriver au résultat r. Les événements sont de la forme $c\,!\,v$ ou $c\,?\,v$, indiquant l'envoi ou la réception

de la valeur v sur le canal identifié par c.

Résultats :	r	::=	v	résultat normal (une valeur)
		$\mid$	`err`	résultat d'erreur
Valeurs :	v	::=	cst	valeur de base
		$\mid$	(v_1, v_2)	paire de valeurs
		$\mid$	(f, x, a, e)	valeur fonctionnelle (fermeture)
		$\mid$	c	identificateur de canal
Environnements :	e	::=	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	
Événements :	evt	::=	$c\,?\,v$	émission d'une valeur
		$\mid$	$c\,!\,v$	réception d'une valeur
Suites d'événements :	w	::=	ε	la suite vide
		$\mid$	$evt \ldots evt$	

On commence par redonner la sémantique des constructions de base, en tenant compte du séquencement des événements extérieurs. Ce séquencement se reflète dans le découpage de la suite w en sous-suites, une pour chaque étape de l'évaluation. Constantes et variables :

$$e \vdash cst \stackrel{\varepsilon}{\Longrightarrow} cst \qquad \frac{x \in \mathrm{Dom}(e)}{e \vdash x \stackrel{\varepsilon}{\Longrightarrow} e(x)} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x \stackrel{\varepsilon}{\Longrightarrow} \texttt{err}}$$

Fonctions :

$$e \vdash (f \ \texttt{where} \ f(x) = a) \stackrel{\varepsilon}{\Longrightarrow} (f, x, a, e)$$

Applications :

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \stackrel{w_3}{\Longrightarrow} r_0}{e \vdash a_1(a_2) \stackrel{w_1 w_2 w_3}{\Longrightarrow} r_0}$$

$$\frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} r_1 \qquad r_1 \text{ n'est pas de la forme } (f, x, a_0, e_0)}{e \vdash a_1(a_2) \stackrel{w}{\Longrightarrow} \texttt{err}}$$

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} \texttt{err}}{e \vdash a_1(a_2) \stackrel{w_1 w_2}{\Longrightarrow} \texttt{err}}$$

Liaisons `let` :

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} v_1 \qquad e + x \mapsto v_1 \vdash a_2 \stackrel{w_2}{\Longrightarrow} r_2}{e \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 \stackrel{w_1 w_2}{\Longrightarrow} r_2} \qquad \frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} \texttt{err}}{e \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 \stackrel{w}{\Longrightarrow} \texttt{err}}$$

Paires :

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} v_1 \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} v_2}{e \vdash (a_1, a_2) \stackrel{w_1 w_2}{\Longrightarrow} (v_1, v_2)} \qquad \frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} \texttt{err}}{e \vdash (a_1, a_2) \stackrel{w}{\Longrightarrow} \texttt{err}} \qquad \frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} v_1 \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} \texttt{err}}{e \vdash (a_1, a_2) \stackrel{w_1 w_2}{\Longrightarrow} \texttt{err}}$$

On décrit maintenant l'évaluation des primitives de parallélisme et de communication, en commençant par la création de canaux.

$$\frac{c \text{ n'est pas alloué ailleurs dans la dérivation}}{e \vdash \texttt{newchan}(a) \stackrel{\varepsilon}{\Longrightarrow} c}$$

Dans la règle d'évaluation de `newchan`, le canal c renvoyé en résultat doit être distinct de tous les autres canaux en service dans le même processus, et dans tous les autres processus se déroulant en parallèle. Pour assurer cette condition au niveau des règles, il faut munir le prédicat d'évaluation de deux arguments supplémentaires : l'ensemble des canaux libres avant et après l'évaluation. Ceci complique les règles au point de les rendre inutilisables. On va s'en tirer en ajoutant une condition au niveau des dérivations : dans toute évaluation considérée par la suite, on exige que deux applications de la règle pour `newchan` assignent toujours des identificateurs différents à c.

Les règles pour l'émission et la réception sont les seules à ajouter des événements $c\,!\,v$ et $c\,?\,v$ dans les séquences d'événements :

$$\frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} c}{e \vdash a_1? \stackrel{w.(c?v)}{\Longrightarrow} v} \qquad \frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} r \qquad r \text{ n'est pas de la forme } c}{e \vdash a_1? \stackrel{w}{\Longrightarrow} \texttt{err}}$$

$$\frac{e \vdash a \stackrel{w}{\Longrightarrow} (c, v)}{e \vdash !(a) \stackrel{w.(c!v)}{\Longrightarrow} \texttt{()}} \qquad \frac{e \vdash a \stackrel{w}{\Longrightarrow} r \qquad r \text{ n'est pas de la forme } (c, v)}{e \vdash !(a) \stackrel{w}{\Longrightarrow} \texttt{err}}$$

L'opération $\oplus$ est bien le choix non-déterministe :

$$\frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} r_1}{e \vdash a_1 \oplus a_2 \stackrel{w}{\Longrightarrow} r_1} \qquad \frac{e \vdash a_2 \stackrel{w}{\Longrightarrow} r_2}{e \vdash a_1 \oplus a_2 \stackrel{w}{\Longrightarrow} r_2}$$

Restent les règles pour la mise en parallèle de deux évaluations :

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} v_1 \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} v_2 \qquad \vdash w_1 \parallel w_2 \Rightarrow w}{e \vdash a_1 \parallel a_2 \stackrel{w}{\Longrightarrow} (v_1, v_2)}$$

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} \texttt{err} \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} r_2 \qquad \vdash w_1 \parallel w_2 \Rightarrow w}{e \vdash a_1 \parallel a_2 \stackrel{w}{\Longrightarrow} \texttt{err}}$$

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} r_1 \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} \texttt{err} \qquad \vdash w_1 \parallel w_2 \Rightarrow w}{e \vdash a_1 \parallel a_2 \stackrel{w}{\Longrightarrow} \texttt{err}}$$

Ces règles utilisent la notion d'entrelacement de deux séquences d'événements. On note $\vdash w_1 \parallel w_2 \Rightarrow w$ pour dire que w est un entrelacement possible de w_1 et de w_2. Cette relation est définie par le système de règles ci-dessous.

$$\vdash \varepsilon \parallel \varepsilon \Rightarrow \varepsilon \qquad \frac{\vdash w_1 \parallel w_2 \Rightarrow w}{\vdash w_1.evt \parallel w_2 \Rightarrow w.evt} \qquad \frac{\vdash w_1 \parallel w_2 \Rightarrow w}{\vdash w_1 \parallel w_2.evt \Rightarrow w.evt}$$

$$\frac{\vdash w_1 \parallel w_2 \Rightarrow w}{\vdash w_1.(c\,?\,v) \parallel w_2.(c\,!\,v) \Rightarrow w} \qquad \frac{\vdash w_1 \parallel w_2 \Rightarrow w}{\vdash w_1.(c\,!\,v) \parallel w_2.(c\,?\,v) \Rightarrow w}$$

Les trois premiers cas d'entrelacement sont triviaux. Les deux dernières règles correspondent au cas d'un rendez-vous réussi entre deux processus évalués en parallèle. Il y a alors effectivement passage de la valeur v d'un processus à l'autre, à travers c. Qui plus est, les deux événements $c\,!\,v$ et $c\,?\,v$ restent internes aux deux processus ; ils ne figurent donc pas dans la séquence d'événements w, qui représente l'activité de communication de $a_1 \parallel a_2$ avec l'extérieur.

Remarque. L'opération d'entrelacement n'est pas déterministe, puisqu'on peut très bien choisir de ne pas effectuer un rendez-vous possible. □

Contexte. En CCS, cette opération d'entrelacement de suites n'est pas explicite : le prédicat de réduction considéré a la forme $a \xrightarrow{\alpha} a'$, où α représente zéro ou un événement ; comme le résultat a' est une autre expression, et non pas une valeur, on peut à nouveau réduire à partir de a', et arriver ainsi à une forme normale en enchaînant les réductions. Dans tout ce travail, j'ai choisi de garder une distinction nette entre programme source et valeur résultat. (En pratique, aucun interpréteur ne procède par réécritures successives du texte source.) D'où, pour le calcul avec communications, les séquences d'événements. □

Exemple. Voici une évaluation possible de $\mathtt{x?} \parallel (\mathtt{x}\,!\,1 \oplus \mathtt{x}\,!\,2)$, dans un environnement e où l'identificateur $\mathtt{x}$ est lié au canal c.

$$\dfrac{\dfrac{e \vdash \mathtt{x} \overset{\varepsilon}{\Rightarrow} c}{e \vdash \mathtt{x?} \overset{c?2}{\Longrightarrow} 2} \qquad \dfrac{\dfrac{e \vdash \mathtt{x} \overset{\varepsilon}{\Rightarrow} c \qquad e \vdash 2 \overset{\varepsilon}{\Rightarrow} 2}{e \vdash (\mathtt{x}, 2) \overset{\varepsilon}{\Rightarrow} (c, 2)}}{e \vdash \mathtt{x}\,!\,2 \overset{c!2}{\Longrightarrow} ()} \qquad \vdash c\,?\,2 \parallel c\,!\,2 \Rightarrow \varepsilon}{e \vdash \mathtt{x?} \parallel (\mathtt{x}\,!\,1 \oplus \mathtt{x}\,!\,2) \overset{\varepsilon}{\Rightarrow} (2, ())}$$

□

2.2.3 Typage

Les canaux de communication se prêtent naturellement à un typage très proche de celui des références. On impose aux canaux d'être homogènes : toutes les valeurs échangées au travers d'un canal donné doivent être de même type. On se convainc aisément, par analogie avec le cas des listes, que le typage statique est extrêmement difficile, voire impossible, sans cette restriction. On donne alors le type τ `chan` aux canaux qui transportent des valeurs de type τ.

$$\begin{array}{rcll} \tau & ::= & \ldots & \\ & | & \tau\ \mathtt{chan} & \text{type d'un canal} \end{array}$$

Pour le typage de la lecture et de l'écriture, on exige que le type de la valeur lue ou écrite soit bien le type des valeurs transmissibles par le canal. On prend donc :

$$\begin{aligned} \texttt{TypOp}(\texttt{newchan}) &= \forall\alpha.\ \texttt{unit} \to \alpha\ \texttt{chan} \\ \texttt{TypOp}(?) &= \forall\alpha.\ \alpha\ \texttt{chan} \to \alpha \\ \texttt{TypOp}(!) &= \forall\alpha.\ \alpha\ \texttt{chan} \times \alpha \to \texttt{unit} \end{aligned}$$

Les constructions $\|$ et $\oplus$ se typent évidemment de la manière suivante, vu leur sémantique :

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1 \parallel a_2 : \tau_1 \times \tau_2} \qquad\qquad \frac{E \vdash a_1 : \tau \qquad E \vdash a_2 : \tau}{E \vdash a_1 \oplus a_2 : \tau}$$

Exemple. La fonction `stamp` plus haut a le type $\forall\alpha, \beta.\ \texttt{int chan} \to \alpha\ \texttt{chan} \to \beta$. □

Contexte. Le typage proposé ci-dessus dépend de manière cruciale du fait que les canaux sont des objets de première classe. Par exemple, dans les calculs de Berthomieu [10] et de Nielson [68], où les processus sont des valeurs, mais pas les canaux, on permet aux canaux de transporter des valeurs de différents types. Le type d'un processus est constitué de l'ensemble des noms de canaux sur lesquels il peut communiquer, avec pour chaque canal le type de la valeur transmise. On notera l'analogie avec le typage des enregistrements (*records*) extensibles [97, 69, 39, 76, 13, 77, 78] : les canaux correspondent aux étiquettes (*labels*) ; les processus, aux enregistrements ; les valeurs transmises, aux valeurs contenues dans les champs. Ce genre de typage ne semble pas s'étendre au cas où les canaux (ou les étiquettes) sont des valeurs : mon impression est que des schémas complexes de types dépendants apparaissent très vite. L'hypothèse d'homogénéité des canaux se justifie alors, et mène au typage très simple donné ici. □

Comme dans le cas des références, il se trouve que le typage des canaux donné ci-dessus est correct tant qu'on ne considère que des canaux de types monomorphes (sans variables de types) ; mais il est incorrect dès qu'on fait intervenir le polymorphisme.

Exemple. Le programme suivant est bien typé :

$$\texttt{let c} = \texttt{newchan}(\texttt{()})\ \texttt{in}\ (\texttt{c}\,!\,\texttt{true}) \parallel (\texttt{1} + \texttt{c}?)$$

On peut en effet donner à `c` le type $\forall\alpha.\ \alpha\ \texttt{chan}$, et donc considérer `c` une fois avec le type `bool chan`, une fois avec le type `int chan`. Ce programme produit évidemment une erreur de type à l'exécution, puisqu'il finit par additionner 1 à `true`. □

2.3 Les continuations

2.3.1 Présentation

Étant donné un programme a_0 et une occurrence d'une sous-expression a dans a_0, on appelle continuation de a (sous-entendu "dans a_0") le calcul qu'il reste à faire, une fois évaluée l'expression a, pour obtenir la valeur de a_0. On peut voir cette continuation comme une fonction qui à la valeur

de a associe la valeur de a_0. Par exemple, dans $a_0 = 2 \times a + 1$, la continuation de a est la fonction $v_a \mapsto 2 \times v_a + 1$. C'est volontairement que je n'ai pas noté $\lambda v_a . 2 \times v_a + 1$; car, dans un langage simple comme celui du chapitre 1, les continuations ne sont pas des objets manipulables au niveau du langage.

La troisième et dernière extension que l'on considère ici consiste précisément à faire des continuations des objets "de première classe", que le programme peut manipuler comme n'importe quelle autre valeur. On fournit de quoi capturer la continuation d'une expression, c'est-à-dire la transformer en un objet continuation. On fournit aussi de quoi abandonner la continuation courante, et installer à sa place le contenu d'un objet continuation. Ceci donne au programmeur toute liberté d'altérer le déroulement normal de l'évaluation : il peut "sauter par-dessus" des calculs, en reprendre d'autres dans un état antérieur, entrelacer des évaluations, ...

Contexte. En fait, presque toutes les structures de contrôle connues — en particulier, les exceptions, les coroutines, les mécanismes de *backtracking*, et même les processus communicants de la partie 2.2 — peuvent se définir dans un langage avec objets continuation : elles n'ont pas besoin d'être primitives [36, 96, 79]. Les objets continuation présentent donc un grand intérêt comme outil de définition de nouvelles structures de contrôles. □

On introduit donc les objets continuation via deux nouvelles opérations primitives :

`Op`	::=	`callcc`	capture de la continuation en cours
	|	`throw`	redémarrage d'une continuation capturée

La primitive `callcc` prend en argument une valeur fonctionnelle. En pratique, on l'emploie uniquement sous la forme `callcc`$(\lambda k.\, a)$. Cette expression lie la variable k à la continuation de l'occurrence de l'expression `callcc`$(\ldots)$ dans le programme tout entier, puis évalue a, et renvoie sa valeur. Quant à la primitive `throw`, elle prend une paire d'arguments : l'objet continuation à relancer, et la valeur à lui fournir.

Exemple. L'expression

$$\texttt{callcc}(\lambda \texttt{k}.\ 1 + (\texttt{if}\ a\ \texttt{then 2 else throw}(\texttt{k}, 10)))$$

s'évalue en 3 si la condition a est vraie, et en 10 si a est fausse. Dans le premier cas, le corps du `callcc` s'évalue en $1 + 2$; cette valeur est aussi celle de l'expression tout entière. Dans le deuxième cas, on redémarre la continuation `k` sur 10. Le calcul en cours, $1 + (\texttt{if} \ldots)$, est interrompu, et on fait comme s'il avait terminé normalement sur la valeur 10, le deuxième argument du `throw`. □

Contexte. Les objets continuation sont d'ordinaire présentés comme des fonctions qui, lorsqu'elles sont appliquées à v, relancent la continuation capturée sur la valeur v. C'est le cas en Scheme [75], par exemple. La présentation ci-dessus, avec une opération spéciale `throw` pour relancer une continuation, est celle de Duba, Harper et MacQueen [25]. Son but premier est de contourner une difficulté de typage des continuations dans le système de Milner. Elle est plus symétrique que la présentation de Scheme, et aussi plus lisible à mon goût. □

Exemple. On considère une présentation simplifiée des exceptions du langage ML, par les deux constructions suivantes : `fail` fait échouer l'évaluation en cours ; a `handle` b évalue a et renvoie

sa valeur, sauf si l'évaluation de a échoue, auquel cas on évalue b à la place. Ces deux opérations se réalisent facilement avec `callcc` et une pile de fonctions. On se donne le type abstrait τ `stack` des piles d'éléments de type τ, modifiables en place. Soit `exn_stack` une variable globale de type (`unit` $\to$ `unit`) `stack`, liée initialement à la pile vide. On traduit alors a `handle` b par :

```
callcc(λk.
    push(exn_stack, λx. throw(k, b));
    let result = a in pop(exn_stack); result)
```

et `fail` par :

```
pop(exn_stack)(());  anything
```

Ici, *anything* est n'importe quelle expression qui est du type τ pour tout τ. Par exemple, on peut prendre pour *anything* la boucle infinie (`f where f(x) = f(x)`)(`0`). Le but de *anything* est d'assurer que `fail` est de type τ pour tout τ. L'expression *anything* n'est de toute façon jamais exécutée. □

Exemple. Avec des continuations, des références, et le cœur bien accroché, on peut programmer toutes sortes de coroutines.

```
callcc(λinit_k.
  let next_k = ref(init_k) in
  let communicate = λx.
    callcc(λk. let old_k = !next_k in next_k := k; throw(old_k, x)) in
  let process1 = f1 where f1(x) =
    ... communicate(y) ... f1(z) ... communicate(w) ... in
  let process2 = f2 where f2(x) =
    ... f2(z) ... communicate(y) ... communicate(t) ... in
  process1(callcc(λstart1.
    process2(callcc(λstart2. next_k := start2; throw(start1, 0))))))
```

Dans l'exemple ci-dessus, les deux fonctions `process1` et `process2` entrelacent leurs calculs *via* la fonction `communicate`. On commence par appeler `process1` sur la valeur initiale 0. Cette fonction exécute une quantité arbitraire de calculs, puis `communicate`(v_1). Le contrôle est alors transféré à `process2` : on commence à évaluer `process2`(v_1). Quand `process2` exécute `communicate`(v_2), on reprend l'évaluation de `process1`. C'est-à-dire, `communicate`(v_1) termine enfin, et renvoie la valeur v_2. Et ainsi de suite. Les fonctions `process1` et `process2` peuvent aussi choisir de terminer normalement ; l'appel correspondant de `communicate` renvoie alors la valeur de retour à l'autre fonction. Tout s'arrête lorsque l'appel initial de `process1` termine. □

2.3.2 Sémantique

L'introduction de `callcc` et de `throw` nécessite des changements de grande envergure dans les règles d'évaluation. Il devient nécessaire d'avoir présent à tout instant un objet sémantique représentant la continuation courante. Le prédicat d'évaluation prend donc maintenant la forme $e \vdash a; k \Rightarrow r$ (lire : "on obtient le résultat r en évaluant l'expression a dans l'environnement e, puis en passant le résultat dans la continuation k"). Ici, k est un terme expliquant ce qu'il reste à faire, une fois évaluée

l'expression a, pour parvenir au résultat r du programme. Les termes de continuations, ainsi que les autres objets sémantiques utilisés, sont définis par la grammaire suivante. (Pour les continuations k, on indique à droite de chaque cas quel instant du calcul est représenté par le terme k.)

Résultats :	r	::=	v	résultat normal (une valeur)
		|	`err`	résultat d'erreur
Valeurs :	v	::=	cst	valeur de base
		|	(v_1, v_2)	paire de valeurs
		|	(f, x, a, e)	valeur fonctionnelle
		|	k	continuation
Environnements :	e	::=	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	
Continuations :	k	::=	`stop`	fin du programme
		|	`primc`(op, k)	après l'argument d'une primitive
		|	`app1c`(a, e, k)	après la partie fonction d'une application
		|	`app2c`(f, x, a, e, k)	après la partie argument d'une application
		|	`letc`(x, a, e, k)	après la partie gauche d'un `let`
		|	`pair1c`(a, e, k)	après le premier argument d'une paire
		|	`pair2c`(v, k)	après le second argument d'une paire

Le constructeur de tête de k décrit ce qu'il faudra faire avec la valeur d'une expression : lui appliquer une primitive (cas `primc`), appeler une fonction (cas `app2c`), évaluer l'autre partie d'un nœud d'application (cas `app1c`), ... Le sous-terme de k qui est une continuation décrit de la même manière les étapes suivantes du calcul.

Il faut bien exécuter un jour les calculs en puissance représentés par k. Cette exécution est décrite par un autre prédicat : $\vdash v \triangleright k \Rightarrow r$ (lire : "la valeur v passée à la continuation k produit la réponse r"). Voici maintenant les règles d'évaluation définissant les deux prédicats $e \vdash a; k \Rightarrow r$ et $\vdash v \triangleright k \Rightarrow r$. Le premier axiome exprime le comportement de la continuation terminale : c'est l'identité.

$$\vdash v \triangleright \texttt{stop} \Rightarrow v$$

Pour les variables, les constantes et les fonctions, la valeur de l'expression se calcule immédiatement, et on la passe aussitôt à la continuation courante.

$$\frac{x \in \mathrm{Dom}(e) \qquad \vdash e(x) \triangleright k \Rightarrow r}{e \vdash x; k \Rightarrow r} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x; k \Rightarrow \texttt{err}}$$

$$\frac{\vdash cst \triangleright k \Rightarrow r}{e \vdash cst; k \Rightarrow r} \qquad \frac{\vdash (f, x, a, e) \triangleright k \Rightarrow r}{e \vdash (f \texttt{ where } f(x) = a); k \Rightarrow r}$$

Pour la liaison `let`, on évalue la première sous-expression en introduisant le constructeur `letc` en tête de la continuation courante. Quand cette évaluation s'achève, elle passe une valeur v à une continuation de la forme `letc`. Ceci provoque l'évaluation de la seconde sous-expression du `let`.

$$\frac{e \vdash a_1; \texttt{letc}(x, a_2, e, k) \Rightarrow r}{e \vdash (\texttt{let } x = a_1 \texttt{ in } a_2); k \Rightarrow r} \qquad \frac{e + x \mapsto v \vdash a_2; k \Rightarrow r}{\vdash v \triangleright \texttt{letc}(x, a_2, e, k) \Rightarrow r}$$

L'évaluation de la paire procède de même, sauf qu'on doit "reprendre la main" deux fois, une fois après l'évaluation de chaque argument.

$$\frac{e \vdash a_1; \texttt{pair1c}(a_2, e, k) \Rightarrow r}{e \vdash (a_1, a_2); k \Rightarrow r} \qquad \frac{e \vdash a_2; \texttt{pair2c}(v_1, k) \Rightarrow r}{\vdash v_1 \triangleright \texttt{pair1c}(a_2, e, k) \Rightarrow r} \qquad \frac{\vdash (v_1, v_2) \triangleright k \Rightarrow r}{\vdash v_2 \triangleright \texttt{pair2c}(v_1, k) \Rightarrow r}$$

Même chose pour l'application de fonction.

$$\frac{e \vdash a_1; \texttt{app1c}(a_2, e, k) \Rightarrow r}{e \vdash a_1(a_2); k \Rightarrow r} \qquad \frac{e_2 \vdash a_2; \texttt{app2c}(f, x, a, e, k) \Rightarrow r}{\vdash (f, x, a, e) \triangleright \texttt{app1c}(a_2, e_2, k) \Rightarrow r}$$

$$\frac{v \text{ n'est pas de la forme } (f, x, a, e)}{\vdash v \triangleright \texttt{app1c}(a_2, e, k) \Rightarrow \texttt{err}} \qquad \frac{e + f \mapsto (f, x, a, e) + x \mapsto v_2 \vdash a; k \Rightarrow r}{\vdash v_2 \triangleright \texttt{app2c}(f, x, a, e, k) \Rightarrow r}$$

Pour toutes les primitives, la première étape est toujours la même : évaluer l'argument.

$$\frac{e \vdash a; \texttt{primc}(op, k) \Rightarrow r}{e \vdash op(a); k \Rightarrow r}$$

La sémantique des primitives est donnée par les règles d'élimination de la continuation `primc`. Pour `callcc`, on duplique la continuation courante k en la stockant dans l'environnement, puis on évalue le corps de la fonction argument comme d'ordinaire.

$$\frac{e + f \mapsto (f, x, a, e) + x \mapsto k \vdash a; k \Rightarrow r}{\vdash (f, x, a, e) \triangleright \texttt{primc}(\texttt{callcc}, k) \Rightarrow r} \qquad \frac{v \text{ n'est pas de la forme } (f, x, a, e)}{\vdash v \triangleright \texttt{primc}(\texttt{callcc}, k) \Rightarrow \texttt{err}}$$

Symétriquement, la règle pour `throw` détruit la continuation courante et utilise à sa place la continuation k' provenant de son argument. (Les évaluations de `callcc` et de `throw` sont les seules qui ne traitent pas la continuation de façon linéaire.)

$$\frac{\vdash v \triangleright k' \Rightarrow r}{\vdash (k', v) \triangleright \texttt{primc}(\texttt{throw}, k) \Rightarrow r} \qquad \frac{v \text{ n'est pas de la forme } (k', v')}{\vdash v \triangleright \texttt{primc}(\texttt{throw}, k) \Rightarrow \texttt{err}}$$

2.3.3 Typage

Suivant une approche éprouvée, on introduit le type τ `cont` des objets continuation qui attendent une valeur de type τ en entrée.

$$\begin{array}{rcll} \tau & ::= & \ldots & \\ & | & \tau \texttt{ cont} & \text{type d'une continuation} \end{array}$$

Le typage de $\texttt{throw}(a_1, a_2)$ consiste alors à s'assurer que a_1 est de type τ `cont` et a_2 de type τ, pour un certain τ. L'expression $\texttt{throw}(a_1, a_2)$ elle-même peut être considérée comme ayant n'importe quel type. En effet, elle ne termine jamais normalement : elle ne renvoie jamais une valeur au contexte

englobant. Aussi le contexte peut-il faire toutes les hypothèses qu'il veut sur le type de la valeur renvoyée. On a donc :

$$\texttt{TypOp}(\texttt{throw}) \quad = \quad \forall\alpha,\beta.\ \alpha\ \texttt{cont} \times \alpha \to \beta$$

Le typage de `callcc` est un peu plus délicat. Considérons le programme $\Gamma[\texttt{callcc}(\lambda k.\, a)]$, où Γ est le contexte de l'expressions `callcc`. Soit τ le type de $\texttt{callcc}(\lambda k.\, a)$. Cette expression renvoie normalement ce que a renvoie ; τ est donc aussi le type de a. Mais τ est aussi le type attendu par le contexte Γ pour la valeur du `callcc`. La continuation k, qui n'est autre qu'une représentation de Γ, attend donc une valeur de type τ ; elle est par conséquent de type $\tau\ \texttt{cont}$. D'où :

$$\texttt{TypOp}(\texttt{callcc}) \quad = \quad \forall\alpha.\ (\alpha\ \texttt{cont} \to \alpha) \to \alpha$$

Contexte. Ce typage est celui proposé par Duba, Harper et MacQueen pour les continuations de SML-NJ [25]. Considérer les objets continuation comme un type abstrait, non comme des fonctions, permet de contourner une des restrictions du système de types de ML (on ne peut quantifier qu'en tête d'un type). Je renvoie le lecteur à l'article de Duba, Harper et MacQueen pour plus de détails, et pour une excellente discussion des autres possibilités de typage. □

Le typage proposé ci-dessus a convaincu tout le monde pendant près de deux ans. Duba, Harper et MacQueen l'ont prouvé correct avec un système de types monomorphes [25] ; ils ont affirmé que leur preuve se généralise sans difficulté à un système de types polymorphes. Mis en confiance, Felleisen et Wright ont publié la correction du typage de `callcc` avec des types polymorphes [100, première édition]. Malheureusement, le typage de `callcc` et de `throw` proposé ci-dessus est incorrect : un objet continuation avec un type polymorphe compromet la sûreté de l'exécution. Voici le premier contre-exemple connu, dû à Harper et Lillibridge [35] :

```
let later =
    callcc(λk.
        (λx. x),
        (λf. throw(k, (f, λx. ()))))
in
    print_string(first(later)("Hello !"));
    second(later)(λx. x + 1)
```

Ce contre-exemple est un peu plus difficile à suivre que ceux pour les références ou les canaux ; mais c'est essentiellement le même phénomène qui se produit. Le typage proposé pour `callcc` et `throw` donne comme type :

$$\texttt{later} \quad : \quad \forall\alpha.\, (\alpha \to \alpha) \times (\alpha \to \alpha) \to \texttt{unit}$$

L'application de `first(later)` à une chaîne de caractères est donc légitime, ainsi que l'application de `second(later)` à la fonction successeur. Pourtant, la continuation k capturée par le `callcc` est λ`later. print_string`... (le corps du `let`). Et donc l'appel à `second(later)` ré-exécute le corps du `let` avec `first(later)` égal à $\lambda\texttt{x.\,x} + 1$, ce qui entraîne une erreur de types à l'exécution lorsqu'on applique `first(later)` à une chaîne de caractères. Comme dans le cas des références et des canaux, on a abusé du fait que la continuation a un type polymorphe pour l'appliquer à des valeurs pas assez générales.

Chapitre 3

Variables dangereuses et types de fermetures

Ce chapitre présente un système de types polymorphes pour les extensions impératives introduites au chapitre 2 (références, canaux, continuations).

3.1 Présentation informelle

Le typage ne garantit plus la sûreté de l'évaluation dès lors qu'il permet les références polymorphes. (Il en va de même pour les canaux polymorphes ou les continuations polymorphes. Pour rester simple, on parle surtout, dans la discussion qui suit, des références ; sauf mention du contraire, tout ce qu'on dit sur les références s'applique également aux canaux et aux continuations.) Il faut donc restreindre le système de types de manière à ce qu'il interdise à une référence d'être considérée avec un type polymorphe — plus précisément, avec un schéma de type non trivial $\forall\alpha.\,\tau[\alpha]$.

3.1.1 Les variables dangereuses

Le système que je propose repose essentiellement sur une restriction de l'étape de généralisation des types (celle qui est effectuée par la construction `let`). On fait en sorte de ne jamais généraliser une variable de type qui est libre dans le type d'une référence accessible. J'appelle ces variables les variables en position dangereuse, ou plus simplement les variables dangereuses. Le type d'une référence peut parfaitement contenir des variables de types ; mais ces variables ne sont jamais généralisables. Ainsi une même référence ne peut-elle être considérée comme appartenant à plusieurs types différents au cours de sa vie.

Il reste maintenant à détecter, au moment de la généralisation, les variables de types qui sont libres dans le type d'une référence accessible. On s'aide ici du système de type lui-même : le principe est que le type de l'expression à généraliser est suffisamment précis pour garder trace des références accessibles depuis la valeur de cette expression. On raisonne par cas sur la nature de l'expression généralisée.

• **Cas d'une référence.** Toutes les variables libres dans son type sont alors dangereuses. Exemple :

```
let r = ref(λx. x) in
    r := (λn. n + 1);
    if (!r)(true) then ... else ...
```

L'expression liée à `r` est du type $(\alpha \to \alpha)$ `ref`. La variable α est en position dangereuse dans ce type : elle apparaît libre sous un constructeur de type `ref`. On ne la généralise donc pas. Le corps du `let` est typé avec `r` : $(\alpha \to \alpha)$ `ref`, et non pas `r` : $\forall\alpha.\,(\alpha \to \alpha)$ `ref`. Le typage de l'affectation n'est possible qu'en instanciant α en `int`. L'application (!`r`)(`true`) est alors mal typée.

• **Cas d'une structure de données.** En ML, le type d'une structure de données (paire, liste, tout type concret) indique non seulement de quel type de structure il s'agit, mais aussi quels sont les types des composantes de la structure. Par exemple, un type liste n'est pas simplement `list`, mais τ `list`, indiquant que toutes les composantes d'une structure de type τ `list` sont de type τ. De même, un type produit $\tau_1 \times \tau_2$ indique que toutes les valeurs de ce type contiennent une composante de type τ_1 et une composante de type τ_2. Les références contenues dans des structures de données laissent donc une trace dans le type de la structure. Ainsi, les variables dangereuses dans τ `list` sont les variables dangereuses dans τ ; de même, les variables dangereuses dans $\tau_1 \times \tau_2$ sont les variables dangereuses dans τ_1 ou dans τ_2. Exemple :

```
let p = (λx. x,  ref(λx. 1)) in a
```

Dans le type $(\alpha \to \alpha) \times ((\beta \to$ `int`$)$ `ref`$)$, la variable β est dangereuse, mais non α. On type donc a sous l'hypothèse `p` : $\forall\alpha.\,(\alpha \to \alpha) \times ((\beta \to$ `int`$)$ `ref`$)$.

Remarque. Ces considérations s'étendent sans difficulté aux types concrets de ML dans toute leur généralité. Soit la déclaration de type concret :

$$\texttt{type}\ (\alpha_1, \ldots \alpha_n)\ T\ =\ C_1\ \texttt{of}\ \tau_1 \mid \ldots \mid C_n\ \texttt{of}\ \tau_n$$

On dit que le $i^{\text{ème}}$ paramètre de T est essentiellement dangereux si α_i apparaît en position dangereuse dans l'un des $\tau_1 \ldots \tau_n$. (Par exemple, si $\tau_i = \alpha_j$ `ref`. Ceci correspond au cas où le type T introduit lui-même une référence.) Dès lors, une variable α est dangereuse dans l'expression de type $(\tau_1, \ldots, \tau_n)\ T$ si α est dangereuse dans un des τ_j, ou bien si α est libre dans τ_i et si le $i^{\text{ème}}$ paramètre de T est essentiellement dangereux. □

• **Cas d'une fonction sans variables libres.** Le cas d'une valeur de type $\tau_1 \to \tau_2$ est spécial. Une valeur de ce type est une fonction qui transforme les valeurs de type τ_1 en valeurs de type τ_2 ; elle ne contient pas en général une composante de type τ_1 ni une composante de type τ_2. Aussi une variable peut-elle être dangereuse dans τ_1 ou dans τ_2, sans pour autant être dangereuse dans $\tau_1 \to \tau_2$. Une variable dangereuse dans τ_1 garde la trace d'une référence dans l'argument qui sera passé à la fonction quand elle sera appliquée ; une variable dangereuse dans τ_2 trahit la présence d'une référence dans le résultat de la fonction, une fois qu'elle aura été appliquée ; mais ces références ne sont pas contenues à l'instant présent dans la valeur fonctionnelle. Exemple :

```
let make_ref = λx. ref(x) in  ...  make_ref(1)  ...  make_ref(true)  ...
```

La variable α n'est pas dangereuse dans $\alpha \to \alpha$ `ref`. Donc, `make_ref` a le type polymorphe $\forall\alpha.\, \alpha \to \alpha$ `ref`, et on peut l'appliquer à des objets de n'importe quel type. Que `make_ref` ait un type aussi général ne menace en rien la sûreté des programmes. En effet, le seul moyen de provoquer une erreur de type à l'exécution avec `make_ref` est de l'appliquer à un objet polymorphe, puis d'utiliser la référence résultat avec deux types différents, via une liaison `let` :

```
let r = make_ref(λx. x) in
    r := (λn. n + 1);
    if (!r)(true) then ... else ...
```

Mais ce dernier programme est mal typé : le résultat de `make_ref` a le type $(\alpha \to \alpha)$ `ref`, et α est dangereuse dans ce type-là, donc `r` reste monomorphe.

3.1.2 Le typage des fermetures

La discussion du cas des fonctions néglige un point crucial : une fonction peut contenir des variables libres, et ces variables libres peuvent être liées à des références. Dans ce cas, la valeur fonctionnelle contient des références, dans la partie environnement de sa fermeture, sans que cela se voie sur son type. Un type fonctionnel $\tau_1 \to \tau_2$ ne dit en effet rien sur le type des variables libres dans les fonctions de ce type.

Exemple. Les références peuvent être présentées sous la forme d'une fonction de lecture et d'une fonction d'écriture.

```
let functional_ref =
    λx. let r = ref(x) in (λ().!r), (λy. r := y) in
let (read, write) =
    functional_ref(λx. x) in
write(λn. n + 1); if read()(true) then ... else ...
```

L'exemple est bien typé, puisqu'on a les types suivants :

$$\begin{aligned} \texttt{functional\_ref} &: \forall\alpha.\, \alpha \to (\texttt{unit} \to \alpha) \times (\alpha \to \texttt{unit}) \\ \texttt{read} &: \forall\beta.\, \texttt{unit} \to (\beta \to \beta) \\ \texttt{write} &: \forall\beta.\, (\beta \to \beta) \to \texttt{unit} \end{aligned}$$

Aucune variable de type n'est dangereuse ici : les types ne contiennent même pas le constructeur `ref`. Et pourtant, le tandem `read`/`write` casse le système de types tout aussi bien qu'une référence polymorphe. □

Que faire ? Mieux typer les fonctions. En effet, les valeurs fonctionnelles sont aussi des structures de données, de composantes les valeurs associées aux variables libres. On le voit bien quand on pense les valeurs fonctionnelles comme des fermetures. Mais, contrairement aux autres structures de données ML, le type d'une valeur fonctionnelle ne permet pas de retrouver les types de ses composantes. On va donc chercher à donner aux valeurs fonctionnelles des types plus détaillés — des types qui typent aussi les fonctions vues comme structures de données. C'est ce que j'appelle le typage des fermetures.

Pour la suite des exemples, on va considérer des types fonctionnels de la forme

$$\tau' \multimap\langle\sigma_1, \ldots, \sigma_n\rangle\rightarrowtail \tau''$$

où les σ_i sont les types des identificateurs libres dans la fonction. (Les σ_i sont en fait des schémas de types, puisque ces identificateurs peuvent avoir des types polymorphes.) Lorsque la fonction est close, on notera souvent $\rightarrow$ au lieu de $\multimap\langle\,\rangle\rightarrowtail$. L'ensemble des σ_i constitue ce que j'appelle un type de fermeture : il type la partie structure de données des valeurs fonctionnelles. Les variables dangereuses dans un tel type fonctionnel sont naturellement les variables dangereuses dans un des σ_i. Dans le dernier exemple, on a donc :

$$\texttt{functional\_ref} : \forall\alpha.\ \alpha \rightarrow (\texttt{unit} \multimap\langle\alpha\ \texttt{ref}\rangle\rightarrowtail \alpha) \times (\alpha \multimap\langle\alpha\ \texttt{ref}\rangle\rightarrowtail \texttt{unit})$$

En effet, les deux fonctions renvoyées contiennent une variable libre, `r`, de type α `ref`. Pour obtenir `read` et `write`, on spécialise α en $\beta \rightarrow \beta$, obtenant les types :

$$\begin{aligned} \texttt{read} &: \texttt{unit} \multimap\langle(\beta \rightarrow \beta)\ \texttt{ref}\rangle\rightarrowtail (\beta \rightarrow \beta) \\ \texttt{write} &: (\beta \rightarrow \beta) \multimap\langle(\beta \rightarrow \beta)\ \texttt{ref}\rangle\rightarrowtail \texttt{unit} \end{aligned}$$

La variable β est dangereuse dans les nouveaux types de `read` et de `write`, puisqu'elle est dangereuse dans le type de fermeture $(\beta \rightarrow \beta)$ `ref`. On ne peut donc pas généraliser β, et l'exemple ne passe pas le typage.

Le typage des fermetures reflète précisément comment une fonction entrelace passages de paramètres et calculs internes, et quelles sont les valeurs qu'elle partage entre plusieurs appels. On a, par exemple :

$$\begin{aligned} \lambda().\texttt{ref}(\lambda\texttt{x}.\,\texttt{x}) &: \texttt{unit} \multimap\langle\,\rangle\rightarrowtail (\alpha \rightarrow \alpha)\ \texttt{ref} \\ \texttt{let r} = \texttt{ref}(\lambda\texttt{x}.\,\texttt{x})\ \texttt{in}\ \lambda().\texttt{r} &: \texttt{unit} \multimap\langle(\alpha \rightarrow \alpha)\ \texttt{ref}\rangle\rightarrowtail (\alpha \rightarrow \alpha)\ \texttt{ref} \end{aligned}$$

Dans le premier cas, α n'est pas dangereuse, et donc généralisable. La fonction renvoie en effet une nouvelle référence à chaque appel. Dans le deuxième cas, α est dangereuse et la fonction reste monomorphe. C'est juste, car la référence `r` est partagée entre tous les appels de la fonction.

3.1.3 Structure des types de fermetures

On m'a souvent demandé pourquoi annoter les types fonctionnels par des ensembles de schémas de types, et non pas par des ensembles de variables dangereuses : les variables dangereuses dans les types des identificateurs libres dans la fonction. Ceci mènerait à des expressions de types plus compactes. Cette approche pose le problème suivant : une fonction peut contenir une variable libre d'un type polymorphe, mais ne contenant pas de variables dangereuses (α, par exemple) ; on ne gardera donc pas trace des variables libres dans ce type ; mais plus tard, ce type polymorphe peut être instancié en un type contenant des variables dangereuses (α devient β `ref`, par exemple). Voici un exemple de cette situation :

```
let K = λx. λy. x in
let f = K(ref(λz. z)) in
    f(0) := λx. x + 1;
    if f(0)(true) then ... else ...
```

La fonction `K` se voit attribuer le type $\forall\alpha,\beta.\ \alpha \to (\beta \to \alpha)$. Les deux flèches ne portent aucune annotation : il est vrai que `x` est libre dans λ`y.x`, mais le type de `x`, qui est α, ne contient aucune variable dangereuse. Dès lors, `f` a le type $\forall\beta,\gamma.\ \beta \to (\gamma \to \gamma)$ `ref`. La variable γ n'est pas dangereuse dans ce type ; on l'a donc généralisée. Et pourtant, le reste de l'exemple provoque une erreur de types à l'exécution. Le problème est que `f` contient une référence vers l'identité ; mais on a perdu trace de cette référence dans les types en la passant à travers `K`.

Une solution est d'annoter les types fonctionnels non seulement par les variables de types qui sont déjà dangereuses dans le type σ d'un identificateur libre, mais aussi par les variables de types qui peuvent devenir dangereuses par instanciation. J'appelle ces variables les variables de types visibles dans le type σ. Toutes les variables libres dans σ ne sont pas forcément visibles ; par exemple, dans $\alpha \to \alpha$, on peut substituer α par n'importe quel type sans que cela ajoute de nouvelles variables dangereuses. Les variables visibles sont, informellement, les variables libres qui apparaissent ailleurs qu'à gauche ou à droite d'une flèche. Dans cette solution, les types fonctionnels sont donc décorés par un ensemble de variables de types marquées ou bien "dangereuses" (notation α `dang`), ou bien "visibles" (notation α `visi`). On obtient par exemple les types :

$$\begin{aligned} \texttt{functional\_ref} &: \forall\alpha.\ \alpha \to (\texttt{unit} \multimap\!\langle \alpha\ \texttt{dang}\rangle\!\rightarrowtail \alpha) \times (\alpha \multimap\!\langle \alpha\ \texttt{dang}\rangle\!\rightarrowtail \texttt{unit}) \\ \texttt{K} &: \forall\alpha,\beta.\ \alpha \to (\beta \multimap\!\langle \alpha\ \texttt{visi}\rangle\!\rightarrowtail \alpha) \end{aligned}$$

En effet, la fonction λ`y.x` possède une variable libre, `x`, de type α ; la variable α est donc marquée visible dans le type de la fonction.

Les règles de substitution sur ces types de fermetures sont inhabituelles. Lorsque α devient τ, le type α `dang` devient α_1 `dang`, $\ldots$, α_n `dang`, où les α_i sont les variables libres dans τ. Quant au type α `visi`, il devient β_1 `dang`, $\ldots$, β_n `dang`, γ_1 `visi`, $\ldots$ γ_m `visi`, où les β_i sont les variables dangereuses dans τ, et les γ_j, les variables visibles mais non dangereuses dans τ. Par exemple, pour appliquer `K` à `ref`(λ`z.z`), on instancie α en $(\gamma \to \gamma)$ `ref` (variables dangereuses : γ ; variables visibles : aucune). Le type du résultat, `f`, est donc :

$$\texttt{f} : \forall\beta.\ \beta \multimap\!\langle \gamma\ \texttt{dang}\rangle\!\rightarrowtail (\gamma \to \gamma)\ \texttt{ref}$$

La variable γ ne peut être généralisée, puisqu'elle est dangereuse dans ce type.

En conclusion : en annotant les types fonctionnels par des ensembles de variables marquées "visibles" ou "dangereuses", on parvient à garder correctement trace des objets dangereux contenus dans des fermetures. On obtient des types de fermetures plus petits, tout en compliquant certaines opérations sur les types de fermetures — tout particulièrement, la substitution. Par la suite, on continuera donc d'annoter les types fonctionnels par des ensembles de schémas de types. Ceci simplifie la formalisation, et fournit une interprétation sémantique plus simple des types de fermetures.

3.1.4 Extensibilité et polymorphisme des types de fermeture

Dans la discussion informelle qui précède, on a annoté les types fonctionnels par des ensembles de schémas de types. Cette vision simple des types de fermetures n'est pas encore satisfaisante.

Premièrement, les types de fermetures doivent être extensibles : il faut qu'un objet de type $\tau \multimap\langle\pi\rangle\rightarrow \tau'$, où π est un ensemble de schémas de types, appartienne aussi au type $\tau \multimap\langle\pi'\rangle\rightarrow \tau'$ pour tout ensemble π' contenant π. C'est sémantiquement correct, puisque tout ce qu'on demande à l'ensemble π, c'est de contenir au moins les types des valeurs contenues dans une fermeture ; π peut très bien contenir des types en plus. Sans ce mécanisme d'extension, le typage des fermetures serait contraignant au point d'être inutilisable. Considérons par exemple deux expressions fonctionnelles avec même type d'argument, même type de résultat, mais des types de fermeture différents :

$$\begin{aligned}(\texttt{f where f(x)} = \ldots) &\;:\; \texttt{int} \multimap\langle \alpha\ \texttt{list}\rangle\rightarrow \texttt{int}\\ (\texttt{g where g(x)} = \ldots) &\;:\; \texttt{int} \multimap\langle \beta\ \texttt{ref}\rangle\rightarrow \texttt{int}\end{aligned}$$

Si les types de fermetures ne sont pas extensibles, on doit rejeter comme mal typée la phrase :

$$\texttt{if ... then (f where f(x)} = \ldots\texttt{) else (g where g(x)} = \ldots\texttt{).}$$

Clairement, cette phrase ne contient pas d'erreur de types. Le typage doit l'accepter. Son type naturel est bien sûr :

$$\texttt{int} \multimap\langle \alpha\ \texttt{list}, \beta\ \texttt{ref}\rangle\rightarrow \texttt{int}$$

En revanche, si les types de fermetures peuvent sont extensibles, le type attendu s'obtient en étendant les types de `f` et de `g` en $\texttt{int} \multimap\langle \alpha\ \texttt{list}, \beta\ \texttt{ref}\rangle\rightarrow \texttt{int}$. Plus généralement, l'extensibilité des types de fermetures doit assurer que deux types fonctionnels sont compatibles si et seulement s'ils ont même type d'argument et même type de résultat — exactement comme dans les systèmes de types habituels.

Deuxième mécanisme qui manque dans le typage des fermetures : avoir des variables de types de fermetures qu'on puisse quantifier universellement, comme on quantifie les variables de types. On veut en effet pouvoir écrire des fonctionnelles qui acceptent en argument toute fonction ayant le bon type d'argument et le bon type de résultat, quel que soit leur type de fermeture. Par exemple, la fonction

$$\texttt{appl where appl(f)} = 2 + \texttt{f(1)}$$

doit s'appliquer à toute valeur de type $\texttt{int} \multimap\langle\pi\rangle\rightarrow \texttt{int}$, quel que soit π. En même temps, il ne faut pas perdre trace des références contenues dans les fermetures passées en arguments aux fonctionnelles. Ceci permettrait de "blanchir" des fonctions contenant des références rien qu'en les passant à travers une fonctionnelle. Exemple :

$$\begin{array}{l}\texttt{let BCCI} = \lambda\texttt{f}.\,\lambda\texttt{x}.\,\texttt{f(x) in}\\ \texttt{let f} = \texttt{let r} = \texttt{ref}(\lambda\texttt{z}.\,\texttt{z})\ \texttt{in}\ \lambda\texttt{y}.\,\texttt{r}\\ \texttt{let g} = \texttt{BCCI(f) in}\\ \quad \texttt{g(1)} := (\lambda\texttt{n}.\,\texttt{n} + 1);\ \ldots\end{array}$$

Supposons qu'on donne le type suivant à la fonctionnelle `BCCI` :

$$\texttt{BCCI} : \forall\alpha, \beta.\, (\alpha \multimap\langle\ \rangle\rightarrow \beta) \rightarrow (\alpha \multimap\left\langle \alpha \multimap\langle\ \rangle\rightarrow \beta\right\rangle\rightarrow \beta)$$

On obtient alors les types suivants pour `f` et `g` :

$$\begin{aligned}\texttt{f} &\;:\; \forall\delta.\, \delta \multimap\left\langle(\gamma \rightarrow \gamma)\ \texttt{ref}\right\rangle\rightarrow (\gamma \rightarrow \gamma)\ \texttt{ref}\\ \texttt{g} &\;:\; \forall\gamma, \delta.\, \delta \multimap\langle\ \rangle\rightarrow (\gamma \rightarrow \gamma)\ \texttt{ref}\end{aligned}$$

La variable γ est dangereuse dans le type de `f`, mais pas dans le type de `g`. En effet, le type de la fermeture de `f` révèle bien la présence d'une référence polymorphe en γ. Mais on a oublié ce fait dans `g`. Et donc, `g`(1) est une référence polymorphe. La source du problème est le type de `BCCI`. Dans ce type, les deux flèches $-\langle\,\rangle\rightarrow$ typent la même fermeture. Aussi, lorsqu'on étend l'un des deux types de fermetures, il faut étendre l'autre de la même manière. Mais ceci n'est pas assuré par le mécanisme d'extension.

Pour résoudre ce problème, on va donc introduire des variables de types de fermetures, notées u, v. Ces variables de types de fermetures servent à représenter les types de fermetures inconnus : ceux des paramètres de fonctions ayant un type fonctionnel. Dès lors, le type correct de `BCCI` est :

$$\texttt{BCCI} : \forall\alpha,\beta,u.\ (\alpha -\langle u\rangle\rightarrow \beta) \rightarrow (\alpha -\Big\langle \alpha -\langle u\rangle\rightarrow \beta\Big\rangle\rightarrow \beta)$$

On généralise une variable de type de fermeture u exactement dans les mêmes circonstances où l'on généralise une variable de type t. On peut instancier u par n'importe quel type de fermeture. L'application `BCCI(f)` est donc bien typée ; u devient γ `list ref`, et on obtient comme type du résultat :

$$\texttt{g}\ \ :\ \ \forall\delta.\ \delta -\Big\langle \delta -\langle \gamma\ \texttt{list ref}\rangle\rightarrow \gamma\ \texttt{list ref}\Big\rangle\rightarrow \gamma\ \texttt{list ref}$$

Ce type garde bien trace de la référence : γ y est dangereuse, et donc non généralisable.

3.2 Un premier système de types

Dans cette partie, on formalise un premier système de types pour les références, les canaux et les continuations, incorporant les idées de variables dangereuses et de typage des fermetures. L'idée maîtresse de ce système est d'utiliser des variables de types de fermetures non seulement pour avoir du polymorphisme sur les types de fermetures, mais aussi pour rendre les types de fermeture extensibles. On va considérer des types de fermetures ayant tous la forme :

$$\pi = \{\sigma_1,\ldots,\sigma_n\}\cup u$$

Un type de fermeture est donc un ensemble de schémas de types, complété d'une variable de type de fermeture u. J'appelle u une variable d'extension ; car il suffit d'instancier u par un type de fermeture $\pi' = \{\sigma'_1,\ldots,\sigma'_m\}\cup u'$ pour obtenir

$$\{\sigma_1,\ldots,\sigma_n,\sigma'_1,\ldots,\sigma'_m\}\cup u',$$

qui est bien une extension du type π.

Contexte. Cette approche consiste en fait à coder avec de la paramétricité une notion de sous-typage (la règle d'extension). Elle s'est révélée fructueuse pour typer de manière polymorphe des objets enregistrement (*records*) extensibles [97, 69, 39, 76, 77, 78, 32]. Les systèmes de type obtenus sont légèrement moins expressifs que les systèmes offrant des mécanismes séparés pour la paramétricité et pour le sous-typage [13], mais ils sont plus simples, et se prêtent mieux à l'inférence de types. Il en va de même dans le cas des types de fermetures. Les problèmes sont cependant bien plus simples que dans le cas des enregistrements : l'intérieur d'un type de fermeture est beaucoup moins structuré que l'intérieur d'un type enregistrement (axiome d'idempotence, absence d'étiquettes). □

Par la suite, on note $\pi = \sigma_1, \ldots, \sigma_n, u$ le type de fermeture $\{\sigma_1, \ldots, \sigma_n\} \cup u$. La virgule peut être vue comme un opérateur binaire associant à droite : l'opérateur qui ajoute un schéma de types à un type de fermeture π.

3.2.1 Expressions de types

On partage les variables de types en deux sortes : les variables de types d'expressions, notées t ; et les variables de types de fermetures, notées u. Une variable de type, notée α ou β, est ou bien une variable de type d'expression, ou bien une variable de type de fermeture.

t	$\in$	`VarTypExp`	variables de types d'expression
u	$\in$	`VarTypClos`	variables de types de fermetures
α, β	$::=$	$t \mid u$	variables de types

On définit trois ensembles de types par la grammaire ci-dessous : l'ensemble `TypExp` des TYPES (plus précisément, les TYPES D'EXPRESSIONS), notés τ ; l'ensemble `TypClos` des TYPES DE FERMETURES, notés π ; et l'ensemble `SchTyp` des SCHÉMAS DE TYPES, notés σ.

τ	$::=$	ι	type de base
	$\mid$	t	variable de type
	$\mid$	$\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$	type fonctionnel
	$\mid$	$\tau_1 \times \tau_2$	type produit
	$\mid$	τ `ref`	type de référence
	$\mid$	τ `chan`	type de canal
	$\mid$	τ `cont`	type de continuation
π	$::=$	u	variable d'extension
	$\mid$	σ, π	ajout du schéma σ au type de fermeture π
σ	$::=$	$\forall\alpha_1 \ldots \alpha_n.\, \tau$	schéma de types

Les substitutions sur cette algèbre de types sont des applications finies des variables de types d'expressions dans les types d'expressions, et des variables de types de fermetures dans les types de fermetures :

$$\text{Substitutions} : \quad \varphi, \psi \quad ::= \quad [t \mapsto \tau, \ldots, u \mapsto \pi, \ldots]$$

De la même manière qu'au chapitre 1, une substitution s'étend naturellement en un homomorphisme de types, de types de fermetures, et de schémas de types. On a en particulier :

$$\begin{aligned} \varphi(\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2) &= \varphi(\tau_1) \multimap\langle\varphi(\pi)\rangle\rightarrow \varphi(\tau_2) \\ \varphi(\sigma, \pi) &= \varphi(\sigma), \varphi(\pi) \end{aligned}$$

On identifie les schémas de types à un renommage près des variables liées par $\forall$:

$$\forall\alpha_1 \ldots \alpha_n.\, \tau = \forall\beta_1 \ldots \beta_n.\, [\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n](\tau).$$

On considère les types de fermetures égaux modulo les deux axiomes suivants :

$$\begin{array}{rcll} \sigma_1, \sigma_2, \pi & = & \sigma_2, \sigma_1, \pi & \text{commutativité à gauche} \\ \sigma, \sigma, \pi & = & \sigma, \pi & \text{idempotence} \end{array}$$

C'est-à-dire, on se place à partir de maintenant dans le quotient de l'ensemble des expressions de types définies ci-dessus par ces deux relations. Par la suite, ce qu'on note τ, σ, π sont des représentants des éléments du quotient. Ces axiomes reflètent la structure d'ensemble (extensible) qu'on veut donner aux types de fermetures. Ils permettent d'identifier un type de fermeture π à un couple (Σ, u) d'un ensemble Σ de schémas et d'une variable d'extension u.

3.2.2 Variables libres, variables dangereuses

A chaque expression de type τ, on associe deux ensembles de variables : $\mathcal{L}(\tau)$, les VARIABLES LIBRES dans τ ; et $\mathcal{D}(\tau)$, les VARIABLES DANGEREUSES dans τ. On définit aussi $\mathcal{L}$ et $\mathcal{D}$ pour les types de fermetures π et pour les schémas de types σ. Voici les définitions, par récurrence sur la syntaxe des expressions :

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{L}(\iota) & = & \emptyset & \mathcal{D}(\iota) & = & \emptyset \\ \mathcal{L}(t) & = & \{t\} & \mathcal{D}(t) & = & \emptyset \\ \mathcal{L}(\tau_1 \multimap\!\langle\pi\rangle\!\rightarrow \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\pi) \cup \mathcal{L}(\tau_2) & \mathcal{D}(\tau_1 \multimap\!\langle\pi\rangle\!\rightarrow \tau_2) & = & \mathcal{D}(\pi) \\ \mathcal{L}(\tau_1 \times \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) & \mathcal{D}(\tau_1 \times \tau_2) & = & \mathcal{D}(\tau_1) \cup \mathcal{D}(\tau_2) \\ \mathcal{L}(\tau\ \texttt{ref}) & = & \mathcal{L}(\tau) & \mathcal{D}(\tau\ \texttt{ref}) & = & \mathcal{L}(\tau) \\ \mathcal{L}(\tau\ \texttt{chan}) & = & \mathcal{L}(\tau) & \mathcal{D}(\tau\ \texttt{chan}) & = & \mathcal{L}(\tau) \\ \mathcal{L}(\tau\ \texttt{cont}) & = & \mathcal{L}(\tau) & \mathcal{D}(\tau\ \texttt{cont}) & = & \mathcal{L}(\tau) \\ \mathcal{L}(u) & = & \{u\} & \mathcal{D}(u) & = & \emptyset \\ \mathcal{L}(\sigma, \pi) & = & \mathcal{L}(\sigma) \cup \mathcal{L}(\pi) & \mathcal{D}(\sigma, \pi) & = & \mathcal{D}(\sigma) \cup \mathcal{D}(\pi) \\ \mathcal{L}(\forall\alpha_1 \ldots \alpha_n.\, \tau) & = & \mathcal{L}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\} & \mathcal{D}(\forall\alpha_1 \ldots \alpha_n.\, \tau) & = & \mathcal{D}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\} \end{array}$$

On voit immédiatement que cette définition de $\mathcal{L}$ et $\mathcal{D}$ passe au quotient par les axiomes d'idempotence et de commutativité gauche.

La proposition suivante montre comment évoluent les variables libres et les variables dangereuses dans un type après substitution.

Proposition 3.1 *Soit φ une substitution. Pour tout type τ, on a :*

$$\begin{array}{rcl} \mathcal{L}(\varphi(\tau)) & = & \left(\bigcup_{\alpha \in \mathcal{L}(\tau)} \mathcal{L}(\varphi(\alpha)) \right) \\ \left(\bigcup_{\alpha \in \mathcal{D}(\tau)} \mathcal{L}(\varphi(\alpha)) \right) \subseteq \mathcal{D}(\varphi(\tau)) & \subseteq & \left(\bigcup_{\alpha \in \mathcal{D}(\tau)} \mathcal{L}(\varphi(\alpha)) \right) \cup \left(\bigcup_{\alpha \in \mathcal{L}(\tau)} \mathcal{D}(\varphi(\alpha)) \right) \end{array}$$

Ces résultats valent aussi pour un type de fermeture π ou un schéma de types σ à la place du type τ.

Démonstration : par récurrence structurelle simultanée sur τ, π et σ. □

3.2.3 Règles de typage

On donne maintenant les règles définissant le prédicat de typage $E \vdash a : \tau$ ("sous les hypothèses E, l'expression a est du type τ"). L'environnement E est une application finie des identificateurs dans les schémas de types. Les seules règles qui diffèrent par-rapport au chapitre 1 sont les règles pour les fonctions et pour le `let`.

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

Comme au chapitre 1, la relation d'instanciation $\leq$ est définie par : $\tau \leq \sigma$ si et seulement si σ est de la forme $\forall\alpha_1 \ldots \alpha_n.\, \tau_0$, et il existe une substitution φ, de domaine inclus dans $\{\alpha_1 \ldots \alpha_n\}$, telle que τ est égal à $\varphi(\tau_0)$.

$$\frac{E + f \mapsto (\tau_1 \multimap\langle E(y_1), \ldots, E(y_n), \pi\rangle\rightarrow \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2 \qquad \{y_1 \ldots y_n\} = \mathcal{I}(f \texttt{ where } f(x) = a)}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle E(y_1), \ldots, E(y_n), \pi\rangle\rightarrow \tau_2}$$

La règle de typage des fonctions exige donc que le type de fermeture de la fonction contienne au moins les types des identificateurs libres dans la fonction. (Même si la règle semble exiger que ces type apparaissent au début du type de fermeture, ils peuvent en fait apparaître n'importe où, en raison de l'axiome de commutativité.)

$$\frac{E \vdash a_1 : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1} \qquad \frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2}$$

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

La règle de typage du `let` est inchangée en elle-même ; ce qui diffère, c'est la définition de l'opérateur `Gen`. On prend maintenant :

$$\texttt{Gen}(\tau, E) = \forall\alpha_1 \ldots \forall\alpha_n.\, \tau \quad \text{avec} \quad \{\alpha_1 \ldots \alpha_n\} = \mathcal{L}(\tau) \setminus \mathcal{D}(\tau) \setminus \mathcal{L}(E).$$

La différence par-rapport à l'opérateur `Gen` utilisé au chapitre 1 est qu'on ne généralise plus les variables de types dangereuses dans τ.

$$\frac{\tau \leq \texttt{TypCst}(cst)}{E \vdash cst : \tau} \qquad \frac{\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2 \leq \texttt{TypOp}(op) \qquad E \vdash a : \tau_1}{E \vdash op(a) : \tau_2}$$

Comme types des opérateurs primitifs sur les références, les canaux, et les continuations, on garde les types naturels introduits au chapitre 2, enrichis de types de fermetures triviaux :

$$\texttt{TypOp(ref)} \quad = \quad \forall t, u.\; t \multimap\langle u\rangle\rightarrow t \texttt{ ref}$$

$$\begin{aligned}
\texttt{TypOp}(!) &= \forall t, u.\ t\ \texttt{ref} \dashv\langle u\rangle\rightarrow t \\
\texttt{TypOp}(:=) &= \forall t, u.\ t\ \texttt{ref} \times t \dashv\langle u\rangle\rightarrow \texttt{unit} \\
\texttt{TypOp}(\texttt{newchan}) &= \forall t, u.\ \texttt{unit} \dashv\langle u\rangle\rightarrow t\ \texttt{chan} \\
\texttt{TypOp}(?) &= \forall t, u.\ t\ \texttt{chan} \dashv\langle u\rangle\rightarrow t \\
\texttt{TypOp}(!) &= \forall t, u.\ t\ \texttt{chan} \times t \dashv\langle u\rangle\rightarrow \texttt{unit} \\
\texttt{TypOp}(\texttt{callcc}) &= \forall t, u, u'.\ (t\ \texttt{cont} \dashv\langle u\rangle\rightarrow t) \dashv\langle u'\rangle\rightarrow t \\
\texttt{TypOp}(\texttt{throw}) &= \forall t, t', u.\ t\ \texttt{cont} \times t \dashv\langle u\rangle\rightarrow t'
\end{aligned}$$

3.2.4 Propriétés du typage

Proposition 3.2 (Stabilité du typage par substitution) *Soient a une expression, τ un type, E un environnement de typage, et φ une substitution. Si $E \vdash a : \tau$, alors $\varphi(E) \vdash a : \varphi(\tau)$.*

Démonstration : par récurrence structurelle sur a. On donne le seul cas qui diffère par-rapport à la preuve de la proposition 1.2.

• **Cas** $a = (\texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2)$**.** La dérivation est de la forme :

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2 : \tau_2}$$

Écrivons $\sigma = \forall\alpha_1 \ldots \forall\alpha_n.\, \tau_1$ avec $\{\alpha_1 \ldots \alpha_n\} = \mathcal{L}(\tau_1) \setminus \mathcal{D}(\tau_1) \setminus \mathcal{L}(E)$. Soient $\beta_1 \ldots \beta_n$ des variables hors de portée de φ et non libres dans E, avec β_i de la même sorte que α_i pour tout i. On note ψ la substitution $\varphi \circ [\alpha_1 \mapsto \beta_1 \ldots \alpha_n \mapsto \beta_n]$.

On applique deux fois l'hypothèse de récurrence : à la prémisse de gauche, avec la substitution ψ ; et à la prémisse de droite, avec la substitution φ. Il vient des preuves de :

$$\psi(E) \vdash a_1 : \psi(\tau_1) \qquad \varphi(E) + x \mapsto \varphi(\texttt{Gen}(\tau_1, E)) \vdash a_2 : \varphi(\tau_2)$$

Puisque les α_i ne sont pas libres dans E, on a $\psi(E) = \varphi(E)$. Il reste maintenant à prouver que $\texttt{Gen}(\psi(\tau_1), \psi(E)) = \varphi(\texttt{Gen}(\tau_1, E))$. Notons

$$V = \mathcal{L}(\psi(\tau_1)) \setminus \mathcal{D}(\psi(\tau_1)) \setminus \mathcal{L}(\psi(E)).$$

Par construction de ψ et des β_i, on a $\psi(\alpha_i) = \varphi(\beta_i) = \beta_i$. De plus, pour toute variable α qui n'est pas une des α_i, aucune des β_i n'est libre dans le terme $\psi(\alpha) = \varphi(\alpha)$.

On fixe i. Comme α_i est libre dans τ_1, on a β_i libre dans $\psi(\tau_1)$ (proposition 3.1, premier résultat). Comme α_i n'est pas libre dans E, on a β_i non libre dans $\psi(E)$. En effet, dans le cas contraire, on aurait $\beta_i \in \mathcal{L}(\psi(\alpha))$ pour un certain α qui de plus appartient à $\mathcal{L}(E)$, par la proposition 3.1. Mais seul α_i peut convenir, et α_i n'est pas libre dans E ; d'où contradiction. Enfin, β_i n'est pas dangereuse dans $\psi(\tau_1)$. En effet, dans le cas contraire, on aurait (par la proposition 3.1, deuxième résultat) soit $\beta_i \in \mathcal{L}(\psi(\alpha))$ pour un certain $\alpha \in \mathcal{D}(\tau_1)$, soit $\beta_i \in \mathcal{D}(\psi(\alpha))$ pour un certain $\alpha \in \mathcal{L}(\tau_1)$. Dans

les deux cas, seul $\alpha = \alpha_i$ peut convenir. Or, α_i n'est pas dans $\mathcal{D}(\tau_1)$, ce qui prouve que la première partie de l'alternative est impossible. Et $\mathcal{D}(\psi(\alpha_i)) = \mathcal{D}(\beta_i) = \emptyset$, ce qui prouve que la deuxième partie de l'alternative est impossible. On conclut donc que

$$\{\beta_1 \dots \beta_n\} \subseteq V.$$

On montre maintenant l'inclusion inverse. Soit β une variable libre dans $\psi(\tau_1)$, et qui n'est pas une des β_i. Soit $\alpha \in \mathcal{L}(\tau_1)$ telle que $\beta \in \mathcal{L}(\psi(\alpha))$. La variable α ne peut être une des α_i, car sinon β serait une des β_i. Donc ou bien α est libre dans E, ou bien α est dangereuse dans τ_1. Si α est libre dans E, alors β est libre dans $\psi(E)$. Si α est dangereuse dans τ_1, alors β est dangereuse dans $\psi(\tau_1)$. Dans les deux cas, $\beta \notin V$. D'où l'inclusion inverse recherchée.

Il s'ensuit que

$$\texttt{Gen}(\psi(\tau_1), \psi(E)) = \forall \beta_1 \dots \beta_n . \psi(\tau_1) = \varphi(\forall \alpha_1 \dots \alpha_n . \tau_1) = \varphi(\texttt{Gen}(\tau_1, E)),$$

par définition de la substitution sur un schéma de types. Et on rappelle que $\psi(E) = \varphi(E)$, puisque les α_i ne sont pas libres dans E. Les deux dérivations obtenues par récurrence permettent donc de conclure, par la règle du `let` :

$$\varphi(E) \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \varphi(\tau_2).$$

C'est le résultat annoncé. □

Proposition 3.3 *Soient a une expression, τ un type, et E, E' deux environnements de typage tels que $\mathrm{Dom}(E) = \mathrm{Dom}(E')$, et $E'(x) \geq E(x)$ pour tout x libre dans a — c'est-à-dire, toute instance de $E(x)$ est aussi instance de $E'(x)$. Si $E \vdash a : \tau$, alors $E' \vdash a : \tau$.*

Démonstration : même preuve que la proposition 1.3. □

3.3 Preuves de sûreté du typage

Dans cette partie, on montre que le système de types proposé ci-dessus est sûr pour chacune des trois sémantiques données au chapitre 2 : celle avec les références, celle avec les canaux, et celle avec les continuations. Les trois preuves sont assez proches, dans leur principe, de la preuve donnée au chapitre 1. Il faut cependant à chaque fois adapter les relations de typage sémantique aux nouveaux objets du langage ; puis montrer qu'il est sémantiquement correct de généraliser une variable non dangereuse ; et enfin, prouver par récurrence sur le déroulement de l'évaluation une propriété de sûreté à chaque fois un peu différente de celle du chapitre 1.

3.3.1 Références

Pour rendre compte du partage de valeurs (*aliasing*) introduit par les références, on ajoute un outil à la panoplie sémantique : les TYPES D'ÉTATS MÉMOIRE (*store typings*). Un type d'état mémoire, noté S, associe une expression de type à chaque adresse mémoire utilisée.

$$\text{Type d'état mémoire :} \quad S \quad ::= \quad [\ell_1 \mapsto \tau_1, \dots, \ell_n \mapsto \tau_n]$$

Le but des types d'états mémoire est d'assurer que toutes les références à l'adresse mémoire ℓ ont le même type monomorphe $S(\ell)$ `ref`, écartant ainsi toute possibilité d'utilisation incohérente de l'adresse ℓ [21, 91, 92]. Le type de l'état mémoire apparaît comme paramètre supplémentaire dans les relations de typage sémantique, qui deviennent :

$S \models v : \tau$	v, considérée dans un état mémoire de type S, est une valeur correcte du type τ
$S \models v : \sigma$	v, considérée dans un état mémoire de type S, est une valeur correcte de toutes les instances du schéma σ
$S \models e : E$	les valeurs contenues dans l'environnement d'évaluation e, considérées dans un état mémoire de type S, appartiennent bien aux schémas correspondants dans E
$\models s : S$	l'état mémoire s est bien du type S.

Voici leurs définitions exactes :

- $S \models cst : \texttt{unit}$ si cst est `()`
- $S \models cst : \texttt{int}$ si cst est un entier
- $S \models cst : \texttt{bool}$ si cst est `true` ou `false`
- $S \models (v_1, v_2) : \tau_1 \times \tau_2$ si $S \models v_1 : \tau_1$ et $S \models v_2 : \tau_2$
- $S \models \ell : \tau\ \texttt{ref}$ si $\ell \in \mathrm{Dom}(S)$ et $\tau = S(\ell)$
- $S \models (f, x, a, e) : \tau_1 \multimap\langle\pi\rangle\rightarrowtail \tau_2$ s'il existe un environnement de typage E tel que

$$S \models e : E \quad \text{et} \quad E \vdash (f\ \texttt{where}\ f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrowtail \tau_2$$

- $S \models v : \forall\alpha_1 \ldots \alpha_n.\tau$ si aucune des variables α_i n'appartient à $\mathcal{D}(\tau)$, et si $S \models v : \varphi(\tau)$ pour toute substitution φ de domaine inclus dans $\{\alpha_1 \ldots \alpha_n\}$
- $S \models e : E$ si $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, et pour tout x dans $\mathrm{Dom}(E)$, on a $S \models e(x) : E(x)$
- $\models s : S$ si $\mathrm{Dom}(s) = \mathrm{Dom}(S)$, et pour tout $\ell \in \mathrm{Dom}(s)$, on a $S \models s(\ell) : S(\ell)$.

Remarque. Dans le cas des valeurs fonctionnelles, on peut supposer de plus que

$$\mathrm{Dom}(E) = \mathcal{I}(f\ \texttt{where}\ f(x) = a).$$

En effet, par les règles de typage, $E \vdash (f\ \texttt{where}\ f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrowtail \tau_2$ implique $\mathrm{Dom}(E) \supseteq \mathcal{I}(f\ \texttt{where}\ f(x) = a)$. Par la proposition 3.3, on peut donc remplacer E par la restriction de E aux identificateurs libres dans $(f\ \texttt{where}\ f(x) = a)$. □

Contexte. En introduisant un type d'état mémoire comme paramètre supplémentaire de $\models$, je ne fais que reprendre l'approche de Tofte [91, 92]. La seule différence est que j'ai remplacé la relation quaternaire $s : S \models v : \tau$ utilisée par Tofte par la conjonction de deux relations plus simples, $S \models v : \tau$ et $\models s : S$. En d'autres termes, pour vérifier que v appartient bien au type τ dans l'état mémoire s et le type mémoire S, je montre dans un premier temps que v a le type τ en admettant

que les valeurs contenues dans les adresses mémoires $\ell_1, \ldots, \ell_n$ accessibles depuis v appartiennent bien aux types $S(\ell_1), \ldots, S(\ell_n)$. Dans un deuxième temps, je vérifie que c'est bien le cas en montrant $S \models s(\ell) : S(\ell)$ pour toutes les adresses ℓ.

La relation quaternaire de Tofte est plus synthétique que ma conjonction de relations, mais considérablement plus difficile à manier. Elle est définie de manière exactement analogue à ma relation $S \models v : \tau$, sauf dans le cas où v est une adresse ℓ. Dans ce cas, Tofte prend $s : S \models \ell : \tau$ `ref` si $S(\ell) = \tau$ et si $s : S \models s(\ell) : \tau$. C'est ce cas qui pose problème : $s(\ell)$ peut être une valeur arbitrairement grande, et donc la définition du prédicat quaternaire n'est pas bien fondée par récurrence sur v. L'état mémoire peut en effet être cyclique, comme dans l'exemple suivant :

```
let r =
  ref(λn. n + 1) in
let fact =
  λn. if n = 0 then 1 else n × !r(n − 1) in
r := fact;  a
```

Au moment où a est évaluée, l'adresse mémoire ℓ à laquelle `r` est liée contient la fermeture de `fact`, et dans la partie environnement de cette fermeture apparaît cette même adresse mémoire ℓ.

Ceci mène Tofte à considérer la pseudo définition récursive du prédicat $s : S \models v : \tau$ comme une équation de point fixe dont la relation $\models$ est solution. Tofte montre que prendre pour $\models$ le plus petit point fixe ne convient pas : dans l'exemple ci-dessus, la valeur de `fact` n'appartient pas sémantiquement au type `int` $\rightarrow$ `int` avec cette définition. Tofte utilise donc le plus grand point fixe, qui se trouve avoir toutes les bonnes propriétés. Malheureusement, les techniques de preuves usuelles par induction ne s'appliquent pas à une relation définie par plus grand point fixe ; il faut faire toutes les preuves par co-induction [62, 91, 92].

Toutes ces complications ne sont pas nécessaires pour ma preuve de sûreté : elle passe très bien en remplaçant le prédicat $s : S \models v : \tau$ par la conjonction de $S \models v : \tau$ et $\models s : S$. (Cette conjonction est plus forte que le prédicat quaternaire : elle exige que toutes les cases mémoires contiennent des valeurs du bon type, alors que la condition $s : S \models v : \tau$ ne s'intéresse qu'aux cases mémoires accessibles depuis la valeur v.) La définition récursive de la relation $S \models v : \tau$ par récurrence sur v est bien fondée, puisque le cas où v est une adresse mémoire est un cas de base. □

On dit qu'un typage de la mémoire S' PROLONGE un autre typage de la mémoire S si $\mathrm{Dom}(S) \subseteq \mathrm{Dom}(S')$, et $S(\ell) = S'(\ell)$ pour tout $\ell \in \mathrm{Dom}(S)$. Cette notion capture un des aspects du déroulement normal d'un programme : de plus en plus d'adresses sont allouées, mais une adresse donnée est toujours considérée avec le même type tout au long ; on peut donc associer à l'exécution une suite croissante de typages de la mémoire, au sens de l'ordre de prolongement. Dans ces circonstances, une relation de typage sémantique comme $S \models v : \tau$ qui est vraie à un instant donné de l'évaluation reste vraie par la suite.

Proposition 3.4 *Si S' prolonge S, alors $S \models v : \tau$ entraîne $S' \models v : \tau$. De même, $S \models e : E$ entraîne $S' \models e : E$.*

Démonstration : récurrence facile sur v. □

La proposition suivante est le lemme crucial de la preuve de sûreté du typage. Il démontre que la notion de variable dangereuse joue bien le rôle attendu : il est toujours sémantiquement correct de généraliser des variables non dangereuses dans un type.

Proposition 3.5 *Soient v une valeur, τ un type et S un type d'état mémoire tels que $S \models v : \tau$. Soient $\alpha_1, \ldots, \alpha_n$ des variables de types telles que $\alpha_i \notin \mathcal{D}(\tau)$ pour tout i. Pour toute substitution φ de domaine inclus dans $\{\alpha_1 \ldots \alpha_n\}$, on a $S \models v : \varphi(\tau)$. En conséquence, $S \models v : \forall \alpha_1 \ldots \alpha_n . \tau$.*

Démonstration : on procède par récurrence structurelle sur v.

• **Cas $v = cst$.** Immédiat, puisque τ est clos dans ce cas.

• **Cas $v = (v_1, v_2)$ et $\tau = \tau_1 \times \tau_2$.** Comme $\mathcal{D}(\tau_1 \times \tau_2) = \mathcal{D}(\tau_1) \cup \mathcal{D}(\tau_2)$, on a $\alpha_i \notin \mathcal{D}(\tau_1)$ et $\alpha_i \notin \mathcal{D}(\tau_2)$ pour tout i. Par hypothèse de récurrence, il vient $S \models v_1 : \varphi(\tau_1)$ et $S \models v_2 : \varphi(\tau_2)$. D'où le résultat.

• **Cas $v = \ell$ et $\tau = \tau_1$ `ref`.** Dans ce cas, $\mathcal{D}(\tau) = \mathcal{L}(\tau_1)$. Comme aucune des α_i n'est dangereuse dans τ, on en déduit qu'aucune des α_i n'est libre dans τ_1. D'où $\varphi(\tau_1) = \tau_1 = S(\ell)$, et le résultat.

• **Cas $v = (f, x, a, e)$ et $\tau = \tau_1 \multimap\!\langle \pi \rangle\!\rightarrow \tau_2$.** Soit E un environnement de typage tel que :

$$S \models e : E \quad \text{et} \quad E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\!\langle \pi \rangle\!\rightarrow \tau_2.$$

On suppose de plus que $\mathrm{Dom}(E) = \mathcal{I}(f \texttt{ where } f(x) = a)$, comme expliqué dans la remarque ci-dessus. On va montrer que

$$S \models e : \varphi(E) \quad \text{et} \quad \varphi(E) \vdash (f \texttt{ where } f(x) = a) : \varphi(\tau_1 \multimap\!\langle \pi \rangle\!\rightarrow \tau_2).$$

La deuxième propriété découle du fait que le typage est stable par substitution (proposition 3.2). Montrons la première propriété. Soit y un identificateur du domaine de E. Il faut établir $S \models e(y) : \varphi(E(y))$. On écrit $E(y)$ sous la forme $\forall \beta_1 \ldots \beta_k . \tau'$, avec les β_i choisies hors de portée de φ, et distinctes des α_i. On a donc $\varphi(E)(y) = \forall \beta_1 \ldots \beta_k . \varphi(\tau')$. Il faut donc montrer $S \models e(y) : \psi(\varphi(\tau'))$ pour toute substitution ψ de domaine inclus dans $\{\beta_1, \ldots, \beta_k\}$. Soit ψ une telle substitution. On a, par définition de $\models$ sur les schémas de types, $\models e(y) : \tau'$, et aucune des β_i n'est dangereuse dans τ'.

Considérons la substitution $\psi \circ \varphi$. On a $\mathrm{Dom}(\psi \circ \varphi) \subseteq \{\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_k\}$. Aucune de ces variables n'est dangereuse dans τ'. Les β_i, par définition de $\models$ sur les schémas de types. Les α_i, parce que $\mathcal{D}(\tau) \setminus \{\beta_1 \ldots \beta_k\} = \mathcal{D}(E(y))$ et que les α_i ne sont pas dangereuse dans $E(y)$. En effet, y est libre dans $(f \texttt{ where } f(x) = a)$, et par la règle de typage des fonctions, $E(y)$ apparaît nécessairement dans le type de fermeture π. Donc $\mathcal{D}(E(y)) \subseteq \mathcal{D}(\pi) = \mathcal{D}(\tau_1 \multimap\!\langle \pi \rangle\!\rightarrow \tau_2)$. Il s'ensuit qu'aucune des α_i n'est dangereuse dans $E(y)$.

On peut donc appliquer l'hypothèse de récurrence à la valeur $e(y)$, au type τ', aux variables $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_k$, et à la substitution $\psi \circ \varphi$. Il vient $\models e(y) : \psi(\varphi(\tau'))$. Ceci vaut pour toute substitution ψ sur les β_i. De plus les β_i sont hors de portée de φ ; il s'ensuit qu'aucune des β_i n'est dangereuse dans $\varphi(\tau')$, pas davantage que dans τ'. On conclut que $\models e(y) : \forall \beta_1 \ldots \beta_k . \varphi(\tau')$, c'est-à-dire que $\models e(y) : \varphi(E)(y)$. Ceci pour tout y. D'où $S \models e : \varphi(E)$, et finalement $S \models (f, x, a, e) : \varphi(\tau)$. □

On va maintenant prouver une propriété de sûreté forte pour le calcul avec références, analogue à la propriété 1.6 pour le calcul purement applicatif.

Proposition 3.6 (Sûreté forte pour les références) *Soient a une expression, τ un type, E un environnement de typage, e un environnement d'évaluation, s un état mémoire, S un typage de la mémoire tels que :*

$$E \vdash a : \tau \quad et \quad S \models e : E \quad et \quad \models s : S.$$

S'il existe une réponse r telle que $e \vdash a/s \Rightarrow r$, alors $r \neq \texttt{err}$; au contraire, r est de la forme v/s', et il existe un typage de la mémoire S' tel que :

$$S' \text{ prolonge } S \quad et \quad S' \models v : \tau \quad et \quad \models s' : S'.$$

Démonstration : la preuve procède par récurrence sur la taille de la dérivation d'évaluation. On raisonne par cas sur a, et donc sur la dernière règle utilisée dans la dérivation de typage. Je donne tous les cas par honnêteté intellectuelle ; le seul cas intéressant est celui du `let`, mais il est très simple une fois qu'on a la proposition 3.5.

• **Cas d'une constante.**

$$\frac{\tau \leq \texttt{TypCst}(cst)}{E \vdash cst : \tau}$$

La seule évaluation possible est $e \vdash cst/s \Rightarrow cst/s$. On vérifie $S \models cst : \texttt{TypCst}(cst)$. On conclut avec $S' = S$.

• **Cas d'une variable.**

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

De l'hypothèse $S \models e : E$ s'ensuit $x \in \mathrm{Dom}(e)$ et $S \models e(x) : E(x)$. La seule évaluation possible est $e \vdash x/s \Rightarrow e(x)/s$. Par définition de $\models$ sur les schémas de types, $S \models e(x) : E(x)$ entraîne $S \models e(x) : \tau$. C'est le résultat attendu, en prenant $S' = S$.

• **Cas d'une fonction.**

$$\frac{E + f \mapsto (\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2}$$

La seule évaluation possible est $e \vdash (f \texttt{ where } f(x) = a)/s \Rightarrow (f, x, a, e)/s$. On a bien $S \models (f, x, a, e) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$ par définition de $\models$, en prenant E pour l'environnement de typage requis. $S' = S$ convient.

• **Cas d'une application.**

$$\frac{E \vdash a_1 : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1}$$

On a trois possibilités d'évaluation. La première conclut $r = \texttt{err}$ parce que $e \vdash a_1 \Rightarrow r_1$ et r_1 n'est pas de la forme $(f, x, a_0, e_0)/s$; mais elle est exclue par l'hypothèse de récurrence appliquée à a_1, qui nous dit que $r_1 = v_1/s_1$ et $\models v_1 : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1$, et donc a fortiori v_1 est une fermeture. La deuxième possibilité d'évaluation conclut $r = \texttt{err}$ parce que $e \vdash a_2 \Rightarrow \texttt{err}$; elle est de même exclue par l'hypothèse de récurrence appliquée à a_2. On est donc dans le troisième cas d'évaluation :

$$\frac{e \vdash a_1/s \Rightarrow (f, x, a_0, e_0)/s_1 \qquad e \vdash a_2/s_1 \Rightarrow v_2/s_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0/s_2 \Rightarrow r}{e \vdash a_1(a_2)/s \Rightarrow r}$$

Par hypothèse de récurrence appliquée à a_1, on obtient un typage mémoire S_1 tel que :

$$S_1 \models (f, x, a, e) : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1 \quad \text{et} \quad \models s_1 : S_1 \quad \text{et} \quad S_1 \text{ prolonge } S.$$

Il existe donc E_0 tel que $S_1 \models e_0 : E_0$, et tel que $E_0 \vdash (f \ \texttt{where}\ f(x) = a_0) : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1$. Une seule règle de typage permet de dériver ce dernier résultat ; est donc vraie sa première prémisse :

$$E_0 + f \mapsto (\tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1) + x \mapsto \tau_2 \vdash a_0 : \tau_1.$$

Appliquant de même l'hypothèse de récurrence à a_2, on obtient S_2 tel que

$$S_2 \models v_2 : \tau_2 \quad \text{et} \quad \models s_2 : S_2 \quad \text{et} \quad S_2 \text{ prolonge } S_1.$$

On considère alors les environnements :

$$e_2 = e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \qquad\qquad E_2 = E_0 + f \mapsto (\tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1) + x \mapsto \tau_1$$

On a, par ce qui précède, $S_2 \models e_2 : E_2$. On peut donc appliquer l'hypothèse de récurrence à l'expression a_0, dans les environnements e_2 et E_2, et dans l'état mémoire $s_2 : S_2$. Il vient que r est de la forme v/s', avec, pour un certain S',

$$S' \models v : \tau_1 \quad \text{et} \quad \models s' : S' \quad \text{et} \quad S' \text{ prolonge } S_2.$$

C'est le résultat attendu, puisque a fortiori S' prolonge S.

• **Cas du `let`.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

On a deux possibilités d'évaluation. La première correspond au cas $e \vdash a_1 \Rightarrow \texttt{err}$. Elle est exclue par l'hypothèse de récurrence appliquée à a_1. L'évaluation est donc de la forme :

$$\frac{e \vdash a_1/s \Rightarrow v_1/s_1 \qquad e + x \mapsto v_1 \vdash a_2/s_1 \Rightarrow r}{e \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2)/s \Rightarrow r}$$

Par hypothèse de récurrence appliquée à a_1, on obtient S_1 tel que :

$$S_1 \models v_1 : \tau_1 \quad \text{et} \quad \models s_1 : S_1 \quad \text{et} \quad S_1 \text{ prolonge } S.$$

Par la proposition 3.5, on a $S_1 \models v_1 : \texttt{Gen}(\tau_1, E)$. En effet, l'opérateur `Gen` ne généralise aucune des variables dangereuses dans τ_1. Notant

$$e_1 = e + x \mapsto v_1 \qquad\qquad E_1 = E + x \mapsto \texttt{Gen}(\tau_1, E),$$

on a donc $S_1 \models e_1 : E_1$. On applique l'hypothèse de récurrence à a_2, e_1, E_1, s_1, S_1. Il vient que r est de la forme v_2/s_2, et il existe S_2 tel que

$$S_2 \models v_2 : \tau_2 \quad \text{et} \quad \models s_2 : S_2 \quad \text{et} \quad S_2 \text{ prolonge } S_1.$$

C'est bien le résultat attendu.

• **Cas de la paire.** Même raisonnement que pour l'application.

• **Cas de la primitive `ref`.**

$$\frac{\tau \multimap\langle\pi\rangle\rightarrow \tau\ \texttt{ref} \leq \forall\alpha, u.\, \alpha \multimap\langle u\rangle\rightarrow \alpha\ \texttt{ref} \qquad E \vdash a : \tau}{E \vdash \texttt{ref}(a) : \tau\ \texttt{ref}}$$

La seule possibilité d'évaluation est :

$$\frac{e \vdash a/s \Rightarrow v/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash \texttt{ref}(a)/s \Rightarrow \ell/(s_1 + \ell \mapsto v)}$$

Par l'hypothèse de récurrence appliquée à a, on obtient S_1 tel que

$$S_1 \models v : \tau \quad \text{et} \quad \models s_1 : S_1 \quad \text{et} \quad S_1 \text{ prolonge } S.$$

Posons $S' = S_1 + \ell \mapsto \tau$. Comme $\mathrm{Dom}(s_1) = \mathrm{Dom}(S_1)$, on a $\ell \notin \mathrm{Dom}(S_1)$, et donc S' prolonge S_1, et donc aussi S. On a donc $S' \models v : \tau$, d'où $\models s_1 + \ell \mapsto v : S'$ et $S' \models \ell : \tau\ \texttt{ref}$, ce qui est le résultat attendu.

• **Cas de la primitive !.**

$$\frac{\tau\ \texttt{ref} \multimap\langle\pi\rangle\rightarrow \tau \leq \forall\alpha, u.\, \alpha\ \texttt{ref} \multimap\langle u\rangle\rightarrow \alpha \qquad E \vdash a : \tau\ \texttt{ref}}{E \vdash\ !a : \tau}$$

On a trois possibilités d'évaluation. L'une mène à `err` parce que a s'évalue en une réponse qui n'est pas de la forme ℓ/s'. Elle est exclue par hypothèse de récurrence appliquée à a. La seconde possibilité est de la forme :

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash\ !a/s_0 \Rightarrow \texttt{err}}$$

Par hypothèse de récurrence appliquée à a, on obtient S_1 tel que $S_1 \models \ell : \tau\ \texttt{ref}$, ce qui implique $l \in \mathrm{Dom}(S_1)$, et $\models s_1 : S_1$, ce qui implique $\mathrm{Dom}(s_1) = \mathrm{Dom}(S_1)$. D'où $\ell \in \mathrm{Dom}(s_1)$, et contradiction. Reste donc la troisième possibilité :

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash\ !a/s_0 \Rightarrow s_1(\ell)/s_1}$$

Par l'hypothèse de récurrence appliquée à a, on obtient S_1 tel que

$$S_1 \models \ell : \tau\ \texttt{ref} \quad \text{et} \quad \models s_1 : S_1 \quad \text{et} \quad S_1 \text{ prolonge } S.$$

On a donc $S_1(\ell) = \tau$, d'où $S_1 \models s_1(\ell) : \tau$, et le résultat recherché, avec $S' = S_1$.

• **Cas de la primitive** :=.

$$\frac{\tau\ \texttt{ref} \times \tau \multimap\langle\pi\rangle\rightarrow \texttt{unit} \leq \forall\alpha, u.\ \alpha\ \texttt{ref} \times \alpha \multimap\langle u\rangle\rightarrow \texttt{unit} \qquad E \vdash a : \tau\ \texttt{ref} \times \tau}{E \vdash :=(a) : \texttt{unit}}$$

Comme dans le cas précédent, la seule possibilité d'évaluation est :

$$\frac{e \vdash a/s_0 \Rightarrow (\ell, v)/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash :=(a)/s_0 \Rightarrow \texttt{()}/(s_1 + \ell \mapsto v)}$$

Par l'hypothèse de récurrence appliquée à a, on obtient S_1 tel que :

$$S_1 \models (\ell, v) : \tau\ \texttt{ref} \quad \text{et} \quad \models s_1 : S_1 \quad \text{et} \quad S_1 \text{ prolonge } S.$$

Ceci entraîne $S_1 \models v : \tau$ et $S_1(\ell) = \tau$. Donc $\models s_1 + \ell \mapsto v : S_1$, et bien sûr $S_1 \models \texttt{()} : \texttt{unit}$. D'où le résultat en prenant $S' = S_1$. □

3.3.2 Canaux de communication

La preuve de sûreté du typage, pour le cas des canaux, est très proche de celle pour le cas des références.

On introduit la notion de typage des canaux : un typage des canaux, noté Γ, associe une expression de type à chaque identificateur de canal c.

$$\text{Typage des canaux :} \quad \Gamma \quad ::= \quad [c_1 \mapsto \tau_1, \ldots, c_n \mapsto \tau_n]$$

On emploie les relations de typage sémantique suivantes :

$\Gamma \models v : \tau$	v est une valeur correcte du type τ
$\Gamma \models v : \sigma$	v est une valeur correcte du schéma de types σ
$\Gamma \models e : E$	les valeurs contenues dans l'environnement d'évaluation e appartiennent bien aux schémas correspondants dans E
$\models w :? \Gamma$	les événements de réception (de la forme $c\,?\,v$) contenus dans la séquence d'événements w respectent les types des canaux attribués par Γ
$\models w :! \Gamma$	les événements d'émission (de la forme $c\,!\,v$) contenus dans la séquence d'événements w respectent les types des canaux attribués par Γ

Voici leur définition exacte :

- $\Gamma \models cst : \texttt{unit}$ si cst est `()`

- $\Gamma \models cst : \texttt{int}$ si cst est un entier
- $\Gamma \models cst : \texttt{bool}$ si cst est `true` ou `false`
- $\Gamma \models (v_1, v_2) : \tau_1 \times \tau_2$ si $\Gamma \models v_1 : \tau_1$ et $\Gamma \models v_2 : \tau_2$
- $\Gamma \models c : \tau$ `chan` si $c \in \mathrm{Dom}(\Gamma)$ et $\tau = \Gamma(c)$
- $\Gamma \models (f, x, a, e) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$ s'il existe un environnement de typage E tel que :

$$\Gamma \models e : E \quad \text{et} \quad E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$$

- $\Gamma \models v : \forall\alpha_1 \ldots \alpha_n.\tau$ si aucune des variables α_i n'appartient à $\mathcal{D}(\tau)$, et si $\Gamma \models v : \varphi(\tau)$ pour toute substitution φ de domaine inclus dans $\{\alpha_1, \ldots, \alpha_n\}$
- $\Gamma \models e : E$ si $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, et pour tout $x \in \mathrm{Dom}(E)$, on a $\Gamma \models e(x) : E(x)$
- $\models w :? \Gamma$ si $\Gamma \models v : \Gamma(c)$ pour tout événement de réception $c\,?\,v$ appartenant à la séquence w
- $\models w :! \Gamma$ si $\Gamma \models v : \Gamma(c)$ pour tout événement d'émission $c\,!\,v$ appartenant à la séquence w.

Remarquons que si w est la concaténation $w_1 \ldots w_n$, on a $\models w :! \Gamma$ si et seulement si $\models w_i :! \Gamma$ pour tout i. De même en remplaçant :! par :? dans cette remarque.

Comme dans le cas des références, les variables dangereuses dans un type τ ont une interprétation sémantique très simple : ce sont les variables qui peuvent être libres dans le type d'un canal accessible à partir d'une valeur du type τ. Il s'ensuit qu'il est sémantiquement correct de généraliser des variables non dangereuses dans un type.

Proposition 3.7 *Soient v une valeur, τ un type et Γ un typage des canaux tels que $\Gamma \models v : \tau$. Soient $\alpha_1 \ldots \alpha_n$ des variables de types telles que $\alpha_i \notin \mathcal{D}(\tau)$ pour tout i. Pour toute substitution φ de domaine inclus dans $\{\alpha_1 \ldots \alpha_n\}$, on a $\Gamma \models v : \varphi(\tau)$. En conséquence, $\Gamma \models v : \forall\alpha_1 \ldots \alpha_n.\tau$.*

Démonstration : même preuve que celle de la proposition 3.5. □

Proposition 3.8 (Sûreté faible pour les canaux) *Soient a_0 une expression et τ_0 un type tels que $[\,] \vdash a_0 : \tau_0$. Soit r_0 une réponse telle que $[\,] \vdash a_0 \stackrel{\varepsilon}{\Rightarrow} r_0$. Alors, $r_0 \neq \texttt{err}$.*

On considère ici l'évaluation d'un programme tout entier, ce afin de définir un typage des canaux Γ global à toute l'évaluation du programme. (La méthode consistant à construire Γ incrémentalement à chaque étape de la preuve de sûreté, comme on l'a fait pour le typage de la mémoire S dans le cas des références, ne marche pas bien en présence d'évaluations parallèles.) Informellement, on va construire Γ de la manière suivante : pour chaque utilisation de la règle d'évaluation de `newchan` dans l'évaluation de a_0,

$$\frac{c \text{ n'est pas alloué ailleurs dans la dérivation}}{e \vdash \texttt{newchan}(a) \stackrel{\varepsilon}{\Rightarrow} c}$$

on prend pour $\Gamma(c)$ le type τ tel que τ `chan` est le type attribué à l'expression `newchan`(a) dans le typage de a_0. Cette construction n'est pas assez précise, car une même expression `newchan`(a) peut apparaître plusieurs fois dans le programme a_0, avec des types différents.

Pour rendre cette construction précise, on va jouer sur le fait que l'argument a de `newchan`(a) est un terme quelconque de type `unit`, qui n'est pas évalué. On se donne une famille infinie de constantes O_i, pour tout entier i, du type `unit` :

$$\begin{array}{rcl} \texttt{Cst} & ::= & \ldots \mid O_1 \mid O_2 \mid \ldots \\ \texttt{TypCst}(O_i) & = & \texttt{unit} \end{array}$$

Soit a_0 l'expression close de l'énoncé 3.8. On construit une expression a_0' en remplaçant dans a_0 tous les sous-termes de la forme `newchan`(a) par `newchan`(O_i), où i est choisi de manière à ce que O_i n'apparaisse qu'une seule fois dans a_0. On vérifie facilement que $[\,] \vdash a_0 : \tau_0$ implique $[\,] \vdash a_0' : \tau_0$. (En effet, la constante O_i est bien du type `unit` exigé pour l'argument de `newchan`.) De même, $[\,] \vdash a_0 \stackrel{\varepsilon}{\Longrightarrow} r_0$ implique $[\,] \vdash a_0 \stackrel{\varepsilon}{\Longrightarrow} r_0'$ pour une certaine réponse r_0' égale à r_0 modulo le remplacement, dans les expressions contenues dans les valeurs fonctionnelles, de sous-termes de la forme `newchan`(a) par des sous-termes de la forme `newchan`(a'). (En effet, l'argument a de `newchan`(a) n'est pas évalué, et peut donc être remplacé par O_i sans changer la structure de la dérivation d'évaluation.) En particulier, si on montre que r_0' ne peut pas être `err`, il s'ensuivra que r_0 ne peut pas être `err`.

On s'est donc ramené à démontrer la proposition 3.8 dans le cas restreint où toutes les créations de canaux apparaissant dans a_0 sont de la forme `newchan`(O_i), avec, pour tout i, O_i n'apparaissant qu'une fois dans a_0. On fixe, dans le reste de cette partie, une dérivation $\mathcal{E}$ de l'évaluation $[\,] \vdash a_0 \stackrel{\varepsilon}{\Longrightarrow} r_0$, et une dérivation $\mathcal{T}$ du typage $[\,] \vdash a_0 : \tau_0$.

Vu les règles de typage, chaque occurrence d'un terme a comme sous-terme de a_0 se voit attribuer un et un seul type τ dans la dérivation de typage $\mathcal{T}$. En conséquence, pour chaque sous-terme `newchan`(O_i) de a, la dérivation $\mathcal{T}$ contient une et une seule dérivation qui conclut $E_i \vdash$ `newchan`$(O_i) : \tau_i$ `chan`, pour un certain type τ_i et pour un certain environnement E_i. On construit alors Γ comme le typage des canaux le moins défini vérifiant la condition suivante. On considère toutes les utilisations de la règle d'évaluation de `newchan` dans la dérivation $\mathcal{E}$:

$$\frac{c \text{ n'est pas alloué ailleurs dans } \mathcal{E}}{e \vdash \texttt{newchan}(O_i) \stackrel{\varepsilon}{\Longrightarrow} c}$$

Pour ce canal c, on prend $\Gamma(c)$ égal au type τ_i tel que τ_i `chan` est le type associé à l'expression `newchan`(O_i) dans la dérivation de typage $\mathcal{T}$. Comme on l'a montré plus haut, ce type τ_i est unique. D'autre part, deux applications de la règle d'évaluation de `newchan` ne peuvent se faire avec le même identificateur de canal c. La condition ci-dessus définit donc bien une fonction Γ des identificateurs de canaux dans les types. De plus, cette fonction Γ est telle que si

$$E \vdash \texttt{newchan}(O_i) : \tau \ \texttt{chan} \quad \text{et} \quad e \vdash \texttt{newchan}(O_i) \stackrel{\varepsilon}{\Longrightarrow} c$$

sont conclusions de sous-dérivations de $\mathcal{T}$ et $\mathcal{E}$ respectivement, alors $\tau = \Gamma(c)$.

Voici la proposition de sûreté sur laquelle on va maintenant travailler par récurrence. Elle est plus complexe que dans la cas des références : les conclusions portent non seulement sur la valeur en laquelle s'évalue une expression, mais aussi sur les valeurs émises sur des canaux pendant l'évaluation ; symétriquement, les hypothèses portent non seulement sur l'environnement d'évaluation initial, mais aussi sur les valeurs reçues à travers des canaux pendant l'évaluation.

Proposition 3.9 (Sûreté forte pour les canaux) *Soit $e \vdash a \stackrel{w}{\Longrightarrow} r$ la conclusion d'une sous-dérivation de $\mathcal{E}$, et $E \vdash a : \tau$ la conclusion d'une sous-dérivation de $\mathcal{T}$, pour la même expression a. On suppose $\Gamma \models e : E$.*

1. *Si $\models w :? \Gamma$, alors $r \neq \texttt{err}$. Au contraire, r est une valeur v, qui vérifie $\Gamma \models v : \tau$. De plus, $\models w :! \Gamma$.*
2. *Si $w = w'.c\,!\,v.w''$ et $\models w' :? \Gamma$, alors $\Gamma \models v : \Gamma(c)$.*

La propriété (2) traduit le fait que si l'évaluation de a en arrive au point d'émettre une valeur sur un canal, alors cette valeur est une valeur correcte du type associé au canal — même si l'évaluation de a produit plus tard une erreur de types, c'est-à-dire si la réponse finale r est `err`. Cette propriété est cruciale pour montrer la sûreté de la mise en parallèle de deux expressions.

Démonstration : la preuve procède par récurrence sur la taille de la sous-dérivation d'évaluation. On raisonne par cas sur a, et donc sur la dernière règle utilisée dans la dérivation de typage. Pour les constructions séquentielles, (1) se prouve de la même manière que dans la preuve de la proposition 3.6 ; je me contente donc de prouver (2). Je donne les preuves complètes pour les constructions de communication et de parallélisme.

• **Cas d'une constante, d'une variable, ou d'une fonction.** (1) est omis. (2) est trivialement vraie, parce que nécessairement $w = \varepsilon$.

• **Cas d'une application.** (1) est omis. Pour (2), on considère toutes les possibilités d'évaluation de $a_1(a_2)$, et toutes les possibilités de décomposition de la séquence d'événements $w'.c\,!\,v.w''$. On note toujours w'_1, w'_2, w'_3 des séquences d'événements bien typées en réception : $\models w'_i :? \Gamma$, en conséquence de l'hypothèse $\models w' :? \Gamma$. On note w''_1, w''_2, w''_3 des séquences d'événements sur lesquelles on ne connaît rien.

$$\frac{e \vdash a_1 \stackrel{w'_1.c!v.w''_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w''_2}{\Longrightarrow} v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \stackrel{w''_3}{\Longrightarrow} r_0}{e \vdash a_1(a_2) \stackrel{w'_1.c!v.w''_1.w''_2.w''_3}{\Longrightarrow} r_0}$$

$$\frac{e \vdash a_1 \stackrel{w'_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w'_2.c!v.w''_2}{\Longrightarrow} v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \stackrel{w''_3}{\Longrightarrow} r_0}{e \vdash a_1(a_2) \stackrel{w'_1.w'_2.c!v.w''_2.w''_3}{\Longrightarrow} r_0}$$

$$\frac{e \vdash a_1 \stackrel{w'_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w'_2}{\Longrightarrow} v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \stackrel{w'_3.c!v.w''_3}{\Longrightarrow} r_0}{e \vdash a_1(a_2) \stackrel{w'_1.w'_2.w'_3.c!v.w''_3}{\Longrightarrow} r_0}$$

$$\frac{e \vdash a_1 \xrightarrow{w'.c!v.w''} r_1 \qquad r_1 \text{ n'est pas de la forme } (f, x, a_0, e_0)}{e \vdash a_1(a_2) \xrightarrow{w'.c!v.w''} \texttt{err}}$$

$$\frac{e \vdash a_1 \xrightarrow{w'_1.c!v.w''_1} (f, x, a_0, e_0) \qquad e \vdash a_2 \xrightarrow{w''_2} \texttt{err}}{e \vdash a_1(a_2) \xrightarrow{w'_1.c!v.w''_1.w''_2} \texttt{err}}$$

$$\frac{e \vdash a_1 \xrightarrow{w'_1} (f, x, a_0, e_0) \qquad e \vdash a_2 \xrightarrow{w'_2.c!v.w''_2} \texttt{err}}{e \vdash a_1(a_2) \xrightarrow{w'_1.w'_2.c!v.w''_2} \texttt{err}}$$

Dans tous les cas, on a une sous-évaluation de la forme $e \vdash a_i \xrightarrow{w'_k.c!v.w''_k} r_i$ à laquelle on peut appliquer l'hypothèse de récurrence (2). On obtient $\Gamma \models v : \Gamma(c)$, ce qui est le résultat recherché.

• **Cas du `let`.** (1) découle de la propriété 3.7. (2) est évidente une fois énumérées toutes les possibilités d'évaluation, comme dans le cas de l'application.

• **Cas de la paire.** (1) est omis. (2) vient par énumération des évaluations.

• **Cas de la création d'un canal.** Montrons (1).

$$\frac{\texttt{unit} \multimap\langle\pi\rangle\rightarrow \tau\ \texttt{chan} \leq \forall\alpha, u.\, \texttt{unit} \multimap\langle u\rangle\rightarrow \alpha\ \texttt{chan} \qquad E \vdash O_i : \texttt{unit}}{E \vdash \texttt{newchan}(O_i) : \tau\ \texttt{chan}}$$

La seule évaluation possible est $e \vdash \texttt{newchan}(O_i) \stackrel{\varepsilon}{\Longrightarrow} c$, pour un certain canal c. Par construction de Γ en fonction des dérivations $\mathcal{D}$ et $\mathcal{T}$, on a $\Gamma(c) = \tau$. D'où (1). (2) est trivialement vraie, puisque $w = \varepsilon$.

• **Cas de la réception sur un canal.**

$$\frac{\tau\ \texttt{chan} \multimap\langle\pi\rangle\rightarrow \tau \leq \forall\alpha, u.\, \alpha\ \texttt{chan} \multimap\langle u\rangle\rightarrow \alpha \qquad E \vdash a : \tau\ \texttt{chan}}{E \vdash a? : \tau}$$

Montrons (1). On a deux évaluations possibles. La première :

$$\frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} r \qquad r \text{ n'est pas de la forme } c}{e \vdash a_1? \stackrel{w}{\Longrightarrow} \texttt{err}}$$

est exclue par l'hypothèse de récurrence (1) appliquée à a, puisque si $e \vdash a_1 \stackrel{w}{\Longrightarrow} r$, alors $\Gamma \models r : \tau\ \texttt{chan}$. On est donc dans le cas :

$$\frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} c}{e \vdash a_1? \xrightarrow{w.(c?v)} v}$$

Comme $\models w.(c\,?\,v) :?\,\Gamma$ par hypothèse, on déduit que $\Gamma \models v : \Gamma(c)$. Et comme $\Gamma \models c : \tau\ \texttt{chan}$, on a $\tau = \Gamma(c)$. D'où $\Gamma \models v : \tau$. Quant à $\models w.(c\,?\,v) :!\,\Gamma$, c'est une conséquence immédiate de $\models w :!\,\Gamma$. La propriété (2) est immédiate par examen des deux cas d'évaluation.

• **Cas de l'émission sur un canal.**

$$\frac{\tau\ \texttt{chan} \times \tau \multimap\!\langle\pi\rangle\!\rightarrow \texttt{unit} \leq \forall\alpha, u.\, \alpha\ \texttt{chan} \times \alpha \multimap\!\langle u\rangle\!\rightarrow \texttt{unit} \qquad E \vdash a : \tau\ \texttt{chan} \times \tau}{E \vdash !(a) : \texttt{unit}}$$

Montrons (1). La première évaluation possible :

$$\frac{e \vdash a \stackrel{w}{\Longrightarrow} r \qquad r \text{ n'est pas de la forme } (c, v)}{e \vdash !(a) \stackrel{w}{\Longrightarrow} \texttt{err}}$$

est exclue, car par l'hypothèse de récurrence (1) appliquée à a, on a $\Gamma \models r : \tau\ \texttt{chan} \times \tau$. On est donc dans le cas :

$$\frac{e \vdash a \stackrel{w'}{\Longrightarrow} (c, v)}{e \vdash !(a) \stackrel{w'.(c!v)}{\Longrightarrow} \texttt{unit}}$$

On a $\Gamma \models \texttt{()} : \texttt{unit}$. Par (1) appliquée à a, on a $\Gamma \models (c, v) : \tau\ \texttt{chan} \times \tau$ et $\models w' :!\,\Gamma$. D'où $\Gamma(c) = \tau$ et $\Gamma \models v : \tau$. Il s'ensuit $\models w'.(c\,!\,v) :!\,\Gamma$. D'où (1) pour $!a$.

Montrons (2). Si l'évaluation se fait par la règle

$$\frac{e \vdash a \stackrel{w}{\Longrightarrow} r \qquad r \text{ n'est pas de la forme } (c, v)}{e \vdash !(a) \stackrel{w}{\Longrightarrow} \texttt{err}}$$

la propriété (2) découle de l'hypothèse de récurrence appliquée à a. Si l'évaluation se fait par la règle

$$\frac{e \vdash a \stackrel{w}{\Longrightarrow} (c, v)}{e \vdash !(a) \stackrel{w.(c!v)}{\Longrightarrow} \texttt{unit}}$$

deux cas sont possibles suivant la décomposition de $w.(c\,!\,v)$ en $w'.(c'\,!\,v').w''$ considérée : ou bien $w'' \neq \varepsilon$, et alors $\Gamma \models v' : \Gamma(c')$ découle du (2) de l'hypothèse de récurrence appliquée à l'évaluation de a ; ou bien $w'' = \varepsilon$ et $w' = w$ et $c' = c$ et $v' = v$, et dans ce cas on a bien $\Gamma \models v : \Gamma(c)$, comme on l'a montré pour établir (1).

• **Cas du choix non-déterministe.** (1) et (2) sont immédiats par hypothèse de récurrence.

• **Cas de la mise en parallèle.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1 \parallel a_2 : \tau_1 \times \tau_2}$$

Montrons (1). Les trois possibilités d'évaluation sont toutes de la forme :

$$\frac{e \vdash a_1 \xRightarrow{w_1} r_1 \qquad e \vdash a_2 \xRightarrow{w_2} r_2 \qquad \vdash w_1 \parallel w_2 \Rightarrow w}{e \vdash a_1 \parallel a_2 \xRightarrow{w} r}$$

On a besoin de $\models w_1 :? \Gamma$ et $\models w_2 :? \Gamma$ pour appliquer l'hypothèse de récurrence à a_1 et a_2. Ceci n'est pas immédiat, car w_1 et w_2 peuvent contenir des événements internes, qui n'apparaissent pas dans w. Par exemple, on peut avoir :

$$w = \varepsilon \qquad w_1 = c\,!\,\texttt{true}.\,\varepsilon \qquad w_2 = c\,?\,\texttt{true}.\,\varepsilon$$

et rien n'indique immédiatement que $\Gamma(c) = \texttt{bool}$, nécessairement. C'est pourtant vrai, mais il faut faire intervenir la propriété (2) : jamais un programme bien typé n'enverrait `true` sur un canal qui n'est pas un `bool chan`. On formalise cet argument dans la sous-proposition 3.10 qui suit.

Sous-proposition 3.10 *Soient w', w'_1, w'_2 des préfixes gauches de w, w_1, w_2 respectivement. Si $\vdash w'_1 \parallel w'_2 \Rightarrow w'$, alors $\models w'_1 :? \Gamma$ et $\models w'_2 :? \Gamma$.*

Démonstration : de la sous-proposition 3.10. La preuve est par récurrence sur la dérivation de $\vdash w'_1 \parallel w'_2 \Rightarrow w'$. Le cas de base $\vdash \varepsilon \parallel \varepsilon \Rightarrow \varepsilon$ est immédiat. Pour les deux cas suivants :

$$\frac{\vdash w'_1 \parallel w'_2 \Rightarrow w'}{\vdash w'_1.evt \parallel w'_2 \Rightarrow w'.evt} \qquad \frac{\vdash w'_1 \parallel w'_2 \Rightarrow w'}{\vdash w'_1 \parallel w'_2.evt \Rightarrow w'.evt}$$

on sait que $\models w'.evt :? \Gamma$, donc si evt est un événement de réception, il est bien typé. Comme $\vdash w'_1 :? \Gamma$ par hypothèse de récurrence, on conclut $\vdash w'_1.evt :? \Gamma$, et de même pour w'_2.

Reste le cas suivant (et son symétrique) :

$$\frac{\vdash w'_1 \parallel w'_2 \Rightarrow w'}{\vdash w'_1.(c\,?\,v) \parallel w'_2.(c\,!\,v) \Rightarrow w'}$$

Les mots w'_1 et w'_2 sont des préfixes gauches de w_1 et w_2, respectivement. Appliquant l'hypothèse de récurrence (de la sous-proposition 3.10), il vient $\models w'_1 :? \Gamma$ et $\models w'_2 :? \Gamma$. On applique l'hypothèse de récurrence (2) (de la proposition 3.9) à l'évaluation $e \vdash a_2 \xRightarrow{w_2} r_2$ et à la décomposition $w_2 = w'_2.(c\,!\,v).w''_2$. Il vient $\Gamma \models v : \Gamma(c)$. D'où $\models w'_1.(c\,?\,v) :? \Gamma$. L'autre résultat, $\models w'_2.(c\,!\,v) :? \Gamma$ est immédiat, puisqu'équivalent à $\models w'_2 :? \Gamma$. □

On reprend maintenant la preuve de la proposition 3.9. Par la sous-proposition 3.10, on a $\models w_1 :? \Gamma$ et $\models w_2 :? \Gamma$. On peut alors appliquer l'hypothèse de récurrence (1) à a_1 et a_2. Il vient

$$r_1 \neq \texttt{err} \quad \text{et} \quad \Gamma \models r_1 : \tau_1 \quad \text{et} \quad \models w_1 :? \Gamma \quad \text{et} \quad r_2 \neq \texttt{err} \quad \text{et} \quad \Gamma \models r_2 : \tau_2 \quad \text{et} \quad \models w_2 :? \Gamma.$$

L'évaluation est donc de la forme :

$$\frac{e \vdash a_1 \xRightarrow{w_1} v_1 \qquad e \vdash a_2 \xRightarrow{w_2} v_2 \qquad \vdash w_1 \parallel w_2 \Rightarrow w}{e \vdash a_1 \parallel a_2 \xRightarrow{w} (v_1, v_2)}$$

On a immédiatement $\Gamma \models (v_1, v_2) : \tau_1 \times \tau_2$. De même, $\models w :! \Gamma$, puisque tous les événements d'émission apparaissant dans w apparaissent ou bien dans w_1, ou bien dans w_2. D'où la propriété (1) pour a.

Quant à la propriété (2), si w se décompose en $w'.c\,!\,v.w''$ avec $\models w' :? \Gamma$, alors l'événement $c\,!\,v$ se retrouve dans w_1 ou dans w_2. Supposons que c'est dans w_1. Alors, w_1 est de la forme $w_1'.c\,!\,v.w_1''$, avec $\models w_1' :? \Gamma$ par la sous-proposition 3.10. Donc, $\Gamma \models v : \Gamma(c)$ par hypothèse de récurrence (2) appliquée à a_1. □

3.3.3 Continuations

Dans le cas des continuations, on utilise des relations de typage sémantique très proches de celles de la partie 1.4. En particulier, on n'a pas besoin d'un argument supplémentaire tenant compte du partage, comme dans le cas des références ou des canaux. On se contente d'ajouter une relation, $\models k :: \tau$, pour le typage des objets continuation. Les relations utilisées sont donc :

$\models v : \tau$	v est une valeur correcte du type τ
$\models v : \sigma$	v est une valeur correcte du schéma de types σ
$\models e : E$	les valeurs contenues dans l'environnement d'évaluation e appartiennent bien aux schémas correspondants dans E
$\models k :: \tau$	la continuation k accepte toutes les valeurs du type τ

Voici leur définition exacte :

- $\models cst : \texttt{unit}$ si cst est `()`
- $\models cst : \texttt{int}$ si cst est un entier
- $\models cst : \texttt{bool}$ si cst est `true` ou `false`
- $\models (v_1, v_2) : \tau_1 \times \tau_2$ si $\models v_1 : \tau_1$ et $\models v_2 : \tau_2$
- $\models k : \tau$ `cont` si $\models k :: \tau$
- $\models (f, x, a, e) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$ s'il existe un environnement de typage E tel que :

$$\models e : E \quad \text{et} \quad E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$$

- $\models v : \forall\alpha_1 \ldots \alpha_n.\tau$ si aucune des variables α_i n'appartient à $\mathcal{D}(\tau)$, et si $\models v : \varphi(\tau)$ pour toute substitution φ de domaine inclus dans $\{\alpha_1, \ldots, \alpha_n\}$
- $\models e : E$ si $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, et pour tout $x \in \mathrm{Dom}(E)$, on a $\models e(x) : E(x)$
- $\models \texttt{stop} :: \tau$ pour tout type τ
- $\models \texttt{app1c}(a, e, k) :: \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$ s'il existe un environnement de typage E tel que

$$E \vdash a : \tau_1 \quad \text{et} \quad \models e : E \quad \text{et} \quad \models k :: \tau_2$$

- $\models \texttt{app2c}(f, x, a, e, k) :: \tau$ s'il existe un environnement de typage E, un type τ' et un type de fermeture π tels que

$$E \vdash (f \ \texttt{where} \ f(x) = a) : \tau \multimap\langle\pi\rangle\rightarrow \tau' \quad \text{et} \quad \models e : E \quad \text{et} \quad \models k :: \tau'$$

- $\models \texttt{letc}(x, a, e, k) :: \tau$ s'il existe un environnement de typage E et un type τ' tels que

$$E + x \mapsto \texttt{Gen}(\tau, E) \vdash a : \tau' \quad \text{et} \quad \models e : E \quad \text{et} \quad \models k :: \tau'$$

- $\models \texttt{pair1c}(a, e, k) :: \tau$ s'il existe un environnement de typage E et un type τ' tels que

$$E \vdash a : \tau' \quad \text{et} \quad \models e : E \quad \text{et} \quad \models k :: \tau \times \tau'$$

- $\models \texttt{pair2c}(v, k) :: \tau$ s'il existe un type τ' tel que

$$\models v : \tau' \quad \text{et} \quad \models k :: \tau' \times \tau$$

- $\models \texttt{primc}(\texttt{callcc}, k) :: \tau \ \texttt{cont} \multimap\langle\pi\rangle\rightarrow \tau$ si $\models k :: \tau$, quel que soit π
- $\models \texttt{primc}(\texttt{throw}, k) :: \tau \ \texttt{cont} \times \tau$ quel que soit τ.

Contexte. Une définition plus simple de $\models k :: \tau$ serait de dire que pour toute valeur v appartenant au type τ, la continuation k appliquée à v ne s'évalue pas en `err`. Plus formellement, on prendrait $\models k :: \tau$ si pour toute valeur v telle que $\models v : \tau$ et pour toute réponse r telles que $\vdash v \triangleright k \Rightarrow r$, on a $r \neq \texttt{err}$. C'est l'approche suivie par Duba, Harper et MacQueen [25]. Cette définition, plus élégante que la définition donnée ci-dessus, n'est malheureusement pas bien fondée par récurrence sur v, puisqu'on quantifie sur une valeur v arbitraire de type τ. Ce n'est pas un problème dans l'article de Duba, Harper et MacQueen, car ils définissent $\models$ sur les valeurs fonctionnelles par la condition de continuité habituelle (voir partie 1.4.1, premier contexte), et donc leur définition de $\models$ est bien fondée par récurrence sur le type. Mais je dois utiliser la condition de Tofte ("il existe E tel que $E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$ et ..."), et cette condition mène à une définition de $\models$ qui n'est pas bien fondée par récurrence sur le type, puisque E peut être plus grand que le type fonctionnel $\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$, en général.[1] D'où la définition de $\models k :: \tau$ par récurrence sur la structure de k et recours fréquent au prédicat de typage présentée ci-dessus. □

Remarque. Sur cette définition du typage sémantique, on voit très bien le problème des continuations en présence de polymorphisme. Le prédicat $\models k :: \tau$ n'est pas stable par substitution de variables de types dans τ : dans le cas `letc`, à partir de $E + x \mapsto \texttt{Gen}(\tau, E) \vdash a : \tau'$, on ne peut pas en général déduire $\varphi(E) + x \mapsto \texttt{Gen}(\varphi(\tau), \varphi(E)) \vdash a : \varphi(\tau')$.

[1] Dans le système du présent chapitre, il se trouve que la définition de $\models$ est également bien fondée par récurrence sur le type, par la magie du typage des fermetures : les morceaux utiles de E, c'est-à-dire $E(y)$ pour tout y libre dans $(f \ \texttt{where} \ f(x) = a)$, apparaissent nécessairement dans le type de fermeture π, et sont donc sous-termes du type de la fonction, $\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$. Cependant, cette propriété ne s'étend pas aux systèmes des chapitres 4 et 6 ; c'est pourquoi je ne me repose pas sur elle ici.

Exemple. On a

$$\models \texttt{letc(x, x(x), [], stop)} : t \dashv\langle u\rangle\rightarrow t$$

puisque `x(x)` est bien typée sous l'hypothèse $\texttt{x} : \forall t, u.\, t \dashv\langle u\rangle\rightarrow t$, mais on n'a pas

$$\models \texttt{letc(x, x(x), [], stop)} : \texttt{int} \dashv\langle u\rangle\rightarrow \texttt{int},$$

puisque l'auto-application est mal typée sous l'hypothèse $\texttt{x} : \forall u.\, \texttt{int} \dashv\langle u\rangle\rightarrow \texttt{int}$. □

C'est pourquoi il n'est pas sémantiquement correct de généraliser n'importe quelle variable de type, contrairement à ce qui se passe dans le calcul purement applicatif du chapitre 1 (proposition 1.5). □

L'interprétation sémantique des variables dangereuses est ici la suivante : les variables dangereuses dans le type τ d'une valeur v sont les variables qui peuvent être libres dans le type d'un objet continuation accessible à partir de v. En conséquence, il est sémantiquement correct de généraliser des variables non dangereuses dans un type.

Proposition 3.11 *Soient v une valeur et τ un type tels que $\models v : \tau$. Soient $\alpha_1 \dots \alpha_n$ des variables de types telles que $\alpha_i \notin \mathcal{D}(\tau)$ pour tout i. Pour toute substitution φ de domaine inclus dans $\{\alpha_1 \dots \alpha_n\}$, on a $\models v : \varphi(\tau)$. En conséquence, $\models v : \forall\alpha_1 \dots \alpha_n.\, \tau$.*

Démonstration : la preuve est une récurrence structurelle sur v, essentiellement identique à celle de la proposition 3.5. Seul diffère le cas de base où v est une continuation k.

• **Cas $v = k$.** Dans ce cas, τ est nécessairement de la forme τ_1 `cont`. Comme $\mathcal{D}(\tau) = \mathcal{L}(\tau_1)$, on déduit qu'aucune des α_i n'est libre dans τ. D'où $\varphi(\tau) = \tau$, et le résultat. □

Proposition 3.12 (Sûreté faible pour les continuations)

1. *Soient a une expression, τ un type, e un environnement d'évaluation, E un environnement de typage, k une continuation et r une réponse tels que*

   $$E \vdash a : \tau \quad et \quad \models e : E \quad et \quad \models k :: \tau \quad et \quad e \vdash a; k \Rightarrow r.$$

   Alors $r \neq \texttt{err}$.

2. *Soient v une valeur, k une continuation, τ un type et r une réponse tels que*

   $$\models v : \tau \quad et \quad \models k :: \tau \quad et \quad \vdash v \triangleright k \Rightarrow r.$$

   Alors $r \neq \texttt{err}$.

Contexte. Je ne donnerai pas de propriété de sûreté forte ("un programme de type `int` s'évalue en un entier") pour le calcul avec continuations. C'est un problème encore ouvert de démontrer la sûreté forte d'un système de types avec une sémantique à continuations [25]. (Voir [100] pour une preuve de sûreté forte dans le cadre d'une sémantique par réécriture.) Qu'on se rassure : la sûreté forte n'est pas, ici, une étape nécessaire pour prouver la sûreté faible. On peut prouver directement la sûreté faible par récurrence, maintenant qu'on a explicité la continuation courante dans la définition de l'évaluation. Prouver ne serait-ce que la sûreté faible du système de Milner avec une sémantique à continuation est déjà une première. Comme l'écrit Milner lui-même [60] :

> When I was working on the original soundness proof of ML typing, wrt a denotational semantics (using ideals), I tried to get the proof to work using a continuation semantics, having worked it out for a direct semantics. The amusing thing was that the proof didn't work. The annoying part is that I can't find the notes. But the memory I have of it is that it was a real crunch point, and that anyone who cares to try to adapt the original proof to a continuation semantics will run into the same difficulty.

J'attribue la difficulté dont parle Milner au typage sémantique des valeurs fonctionnelles. Étant dans un modèle dénotationnel, il utilisait certainement la condition de continuité habituelle ($f : \tau_1 \to \tau_2$ si pour toute valeur $v : \tau_1$ et pour toute continuation $k :: \tau_2$ on a $f(v)(k) \neq \texttt{wrong}$), qui n'a aucune raison d'être stable par instanciation du type. En revanche, m'étant placé dans un cadre opérationnel, je peux utiliser la condition de Tofte (partie 1.4.1, premier contexte), qui, elle, est stable par instanciation. □

Démonstration : on prouve simultanément (1) et (2) par récurrence sur la taille des dérivations d'évaluation (de $e \vdash a; k \Rightarrow r$ et de $\vdash v \triangleright k \Rightarrow r$ respectivement). On raisonne par cas sur a pour (1), et par cas sur k pour (2).

• **(1), cas d'une constante.**

$$\frac{\tau \leq \texttt{TypCst}(cst)}{E \vdash cst : \tau}$$

La seule évaluation possible est :

$$\frac{\vdash cst \triangleright k \Rightarrow r}{e \vdash cst; k \Rightarrow r}$$

On a $\models cst : \tau$, vu la définition de `TypCst`. Comme $\models k :: \tau$, appliquant l'hypothèse de récurrence (2) à l'évaluation $\vdash cst \triangleright k \Rightarrow r$, il vient $r \neq \texttt{err}$.

• **(1), cas d'une variable.**

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

De l'hypothèse $\models e : E$ s'ensuit $x \in \mathrm{Dom}(e)$ et $\models e(x) : E(x)$. La seule évaluation possible est donc :

$$\frac{x \in \mathrm{Dom}(e) \qquad \vdash e(x) \triangleright k \Rightarrow r}{e \vdash cst; k \Rightarrow r}$$

Comme $\models e(x) : E(x)$, on a $\models e(x) : \tau$, et donc $r \neq \texttt{err}$ d'après (2) et l'hypothèse $\models k :: \tau$.

• **(1), cas d'une fonction.**

$$\frac{E + f \mapsto (\tau_1 \multimap\langle\pi\rangle\rightarrowtail \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrowtail \tau_2}$$

La seule évaluation possible est :

$$\frac{\vdash (f, x, a, e) \triangleright k \Rightarrow r}{e \vdash (f \texttt{ where } f(x) = a); k \Rightarrow r}$$

On a $\models (f, x, a, e) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$ par définition de $\models$ (en prenant E pour l'environnement de typage requis). Puisque $\models k :: \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$, il vient $r \neq \texttt{err}$ par (2).

• **(1), cas d'une application.**

$$\frac{E \vdash a_1 : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1}$$

L'évaluation est de la forme :

$$\frac{e \vdash a_1; \texttt{app1c}(a_2, e, k) \Rightarrow r}{e \vdash a_1(a_2); k \Rightarrow r}$$

On a $\models \texttt{app1c}(a_2, e, k) :: \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1$ par définition de $\models$ sur les continuations `app1c`, en prenant E pour l'environnement de typage requis. Appliquant l'hypothèse de récurrence (1) à la prémisse de la règle d'évaluation, il vient $r \neq \texttt{err}$.

• **(1), cas du `let`.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

L'évaluation est nécessairement de la forme :

$$\frac{e \vdash a_1; \texttt{letc}(x, a_2, e, k) \Rightarrow r}{e \vdash (\texttt{let } x = a_1 \texttt{ in } a_2); k \Rightarrow r}$$

On a $\models \texttt{letc}(x, a_2, e, k) :: \tau_1$ par définition de $\models$ sur les continuations `letc`, en prenant E pour l'environnement requis, et τ_2 pour le type requis. Le résultat $r \neq \texttt{err}$ s'ensuit de l'hypothèse de récurrence (1).

• **(1), cas de la paire.** Même raisonnement que pour l'application.

• **(1), cas de la primitive `callcc`.**

$$\frac{(\tau \texttt{ cont} \multimap\langle\pi'\rangle\rightarrow \tau) \multimap\langle\pi\rangle\rightarrow \tau \leq \forall t, u, v.\, (t \texttt{ cont} \multimap\langle u\rangle\rightarrow t) \multimap\langle v\rangle\rightarrow t \qquad E \vdash a : \tau \texttt{ cont} \multimap\langle\pi'\rangle\rightarrow \tau}{E \vdash \texttt{callcc}(a) : \tau}$$

L'évaluation est de la forme :

$$\frac{e \vdash a; \texttt{primc}(\texttt{callcc}, k) \Rightarrow r}{e \vdash \texttt{callcc}(a); k \Rightarrow r}$$

On a $\models \texttt{primc}(\texttt{callcc}, k) : \tau\ \texttt{cont} \dashv\langle\pi'\rangle\rightarrow \tau$, puisque $\models k :: \tau$. D'où $r \neq \texttt{err}$ par (1).

• **(1), cas de la primitive `throw`.** Même raisonnement que pour `callcc`.

• **(2), cas de la continuation `stop`.** La seule évaluation possible est $\vdash v \triangleright \texttt{stop} \Rightarrow v$, donc r est égale à v, et donc r n'est pas `err`.

• **(2), cas de la continuation `app1c`.** On a k de la forme $\texttt{app1c}(a_1, e_1, k_1)$. Par hypothèse $\models k :: \tau$, le type τ est de la forme $\tau_1 \dashv\langle\pi\rangle\rightarrow \tau_2$, avec

$$E_1 \vdash a_1 : \tau_1 \quad \text{et} \quad \models e_1 : E_1 \quad \text{et} \quad \models k_1 :: \tau_2$$

pour un certain environnement E. Par hypothèse $\models v : \tau$, la valeur v est une fermeture (f, x, a, e), et il existe un environnement de typage E tel que

$$\models e : E \quad (3) \quad \text{et} \quad E \vdash (f\ \texttt{where}\ f(x) = a) : \tau_1 \dashv\langle\pi\rangle\rightarrow \tau_2 \quad (4).$$

Ceci exclut la première possibilité d'évaluation : celle qui conclut $r = \texttt{err}$ parce que v n'est pas une fermeture. On est donc dans l'autre cas d'évaluation :

$$\frac{e_1 \vdash a_1; \texttt{app2c}(f, x, a, e, k) \Rightarrow r}{\vdash (f, x, a, e) \triangleright \texttt{app1c}(a_1, e_1, k_1) \Rightarrow r}$$

De (3) et (4), on tire $\models \texttt{app2c}(f, x, a, e, k) :: \tau_1$ par définition de $\models$ sur les continuations `appc2`. Appliquant l'hypothèse de récurrence (1) à l'évaluation de $e_1 \vdash a_1; \texttt{app2c}(f, x, a, e, k) \Rightarrow r$, il vient $r \neq \texttt{err}$, comme annoncé.

• **(2), cas de la continuation `app2c`.** On a k de la forme $\texttt{app2c}(f, x, a, e, k)$. Par hypothèse $\models k :: \tau$, on a

$$E \vdash (f\ \texttt{where}\ f(x) = a) : \tau \dashv\langle\pi\rangle\rightarrow \tau' \quad (3) \quad \text{et} \quad \models e : E \quad (4) \quad \text{et} \quad \models k :: \tau' \quad (5)$$

pour un certain environnement E, et certains types τ' et π. La seule possibilité d'évaluation est

$$\frac{e + f \mapsto (f, x, a, e) + x \mapsto v_2 \vdash a; k \Rightarrow r}{\vdash v_2 \triangleright \texttt{app2c}(f, x, a, e, k) \Rightarrow r}$$

Considérons les environnements :

$$e_1 = e + f \mapsto (f, x, a, e) + x \mapsto v \quad \text{et} \quad E_1 = e + f \mapsto (\tau \dashv\langle\pi\rangle\rightarrow \tau') + x \mapsto \tau.$$

Par (3) et (4), on a $\models (f, x, a, e) : \tau \dashv\langle\pi\rangle\rightarrow \tau'$. Joint à l'hypothèse $\models v : \tau$ et à (4), ce fait entraîne $\models e_1 : E_1$. D'autre part, une seule règle de typage permet de dériver (3) ; est donc vraie sa prémisse : $E_1 \vdash a : \tau'$. On peut donc appliquer l'hypothèse de récurrence (1) à l'évaluation $e_1 \vdash a; k \Rightarrow r$. Il vient le résultat attendu : $r \neq \texttt{err}$.

• **(2), cas de la continuation `letc`.** La continuation k est de la forme $\texttt{letc}(x, a, e, k')$. Par hypothèse $\models k :: \tau$, on a

$$E + x \mapsto \texttt{Gen}(\tau, E) \vdash a : \tau' \quad \text{et} \quad \models e : E \quad \text{et} \quad \models k' :: \tau'$$

pour un certain environnement E et un certain type τ'. La dernière étape de l'évaluation est nécessairement :

$$\frac{e + x \mapsto v \vdash a; k' \Rightarrow r}{\vdash v \triangleright \texttt{letc}(x, a, e, k') \Rightarrow r}$$

Par hypothèse, on a $\models v : \tau$. Comme `Gen` ne généralise pas de variables dangereuses dans τ, il s'ensuit $\models v : \texttt{Gen}(\tau, E)$ par la proposition 3.11. D'où :

$$\models (e + x \mapsto v) : (E + x \mapsto \texttt{Gen}(\tau, E)).$$

On peut donc appliquer l'hypothèse de récurrence (1) à l'évaluation $e + x \mapsto v \vdash a; k' \Rightarrow r$. Il vient $r \neq \texttt{err}$, ce qui est le résultat attendu.

• **(2), cas de la continuation `pair1c`.** Semblable au cas `app1c`.

• **(2), cas de la continuation `pair2c`.** Immédiat.

• **(2), cas de la continuation $\texttt{primc}(\texttt{callcc}, k')$.** Par hypothèse $\models k :: \tau$, on a τ de la forme $\tau'\ \texttt{cont} \multimap\langle\pi\rangle\rightarrowtail \tau'$, et $\models k' :: \tau'$. Par hypothèse $\models v : \tau$, on a donc, pour un certain E,

$$v = (f, x, a, e) \quad \text{et} \quad \models e : E \quad (3) \quad \text{et} \quad E \vdash (f\ \texttt{where}\ f(x) = a) : \tau'\ \texttt{cont} \multimap\langle\pi\rangle\rightarrowtail \tau' \quad (4)$$

Il y a deux possibilités d'évaluation. La première mène à $r = \texttt{err}$ parce que v n'est pas une fermeture ; elle est exclue. La deuxième se conclut par :

$$\frac{e + f \mapsto (f, x, a, e) + x \mapsto k \vdash a; k \Rightarrow r}{\vdash (f, x, a, e) \triangleright \texttt{primc}(\texttt{callcc}, k) \Rightarrow r}$$

De (3) et des hypothèses sur v et k, il s'ensuit que

$$\models (e + f \mapsto (f, x, a, e) + x \mapsto k) : (E + f \mapsto (\tau'\ \texttt{cont} \multimap\langle\pi\rangle\rightarrowtail \tau') + x \mapsto \tau'\ \texttt{cont}).$$

De (4) et de la règle de typage des fonctions, il vient

$$E + f \mapsto (\tau'\ \texttt{cont} \multimap\langle\pi\rangle\rightarrowtail \tau') + x \mapsto \tau' \vdash a : \tau'.$$

On peut donc appliquer l'hypothèse de récurrence (1) à l'évaluation $e + f \mapsto (f, x, a, e) + x \mapsto k \vdash a; k \Rightarrow r$. Il vient, comme annoncé, $r \neq \texttt{err}$.

• **(2), cas de la continuation $\texttt{primc}(\texttt{throw}, k_1)$.** Comme $\models k :: \tau$, on a τ de la forme $\tau'\ \texttt{cont} \times \tau'$. L'hypothèse $\models v : \tau$ se traduit donc par

$$v = (k', v') \quad \text{et} \quad \models k' :: \tau' \quad \text{et} \quad \models v' : \tau'.$$

Ceci élimine la première possibilité d'évaluation : celle qui conclut $r = \texttt{err}$ parce que v n'est pas de la forme (k', v'). Reste donc la seconde possibilité :

$$\frac{\vdash v' \triangleright k' \Rightarrow r}{\vdash (k', v') \triangleright \texttt{primc}(\texttt{throw}, k_1) \Rightarrow r}$$

On applique l'hypothèse de récurrence (2) à l'évaluation $\vdash v' \triangleright k' \Rightarrow r$. Il vient $r \neq \texttt{err}$. Ceci achève la preuve. □

3.4 Inférence de types

Dans cette partie, on montre que toute expression bien typée possède un type principal, et on donne un algorithme d'inférence de types — une adaptation de celui de Damas et Milner — qui calcule ce type principal.

3.4.1 Problèmes avec l'unification

Le système de types du présent chapitre ne se prête pas de manière immédiate à l'inférence de types. En effet, en raison des axiomes de commutativité et d'idempotence mis sur les types de fermetures, l'algèbre des types ne possède pas la propriété d'unificateur principal.

Exemple. On considère les types

$$\tau_1 = t \multimap\langle \texttt{int}, \texttt{bool}, u\rangle\rightarrow t \qquad \tau_2 = t \multimap\langle \texttt{int}, \texttt{char}, v\rangle\rightarrow t$$

Voici deux unificateurs de τ_1 et τ_2 :

$$\begin{aligned} \varphi_1 &= [u \mapsto \texttt{char}, w;\ v \mapsto \texttt{bool}, w] \\ \varphi_2 &= [u \mapsto \texttt{char}, \texttt{int}, w;\ v \mapsto \texttt{bool}, w] \end{aligned}$$

En effet, on a bien, par idempotence et commutativité gauche :

$$\varphi_2(\texttt{int}, \texttt{bool}, u) = \texttt{int}, \texttt{bool}, \texttt{char}, \texttt{int}, w = \texttt{int}, \texttt{bool}, \texttt{char}, w = \varphi_2(\texttt{int}, \texttt{char}, v).$$

Et pourtant, φ_1 et φ_2 sont incomparables : il n'y a pas de substitution θ telle que $\varphi_1 = \theta \circ \varphi_2$ ou $\varphi_2 = \theta \circ \varphi_1$. □

Exemple. On considère les types

$$\tau_1 = t \multimap\langle t\ \texttt{ref}, u\rangle\rightarrow t \qquad \tau_2 = t \multimap\langle \texttt{int ref}, v\rangle\rightarrow t.$$

Les deux substitutions ci-dessous unifient τ_1 et τ_2 :

$$\varphi_1 = [u \mapsto \texttt{int ref}, w;\ v \mapsto t\ \texttt{ref}, w] \qquad \varphi_2 = [t \mapsto \texttt{int};\ u \mapsto w;\ v \mapsto w].$$

On a en effet

$$\varphi_1(t\ \texttt{ref}, u) = t\ \texttt{ref}, \texttt{int ref}, t\ \texttt{ref}, v = t\ \texttt{ref}, \texttt{int ref}, v = \varphi_1(\texttt{int ref}, u)$$

par idempotence et commutativité. Pourtant, les deux substitutions φ_1 et φ_2 sont incomparables. □

Dans le premier exemple ci-dessus, les deux unificateurs envoient tous deux τ_1 et τ_2 sur $t \multimap\langle \texttt{int}, \texttt{bool}, \texttt{char}, w\rangle\!\rightarrowtail t$, ce qui, intuitivement, est bien l'instance commune la plus générale des deux types. Le seul moyen de distinguer φ_1 et φ_2 est de les appliquer à des types de fermetures qui se terminent par u ; et encore faut-il que ces types ne contiennent pas `int`. En particulier, φ_1 et φ_2 sont équivalentes si, parmi tous les types de fermetures considérés, les seuls qui se terminent par u sont toujours égaux à $\texttt{int}, \texttt{bool}, u$.

Il se trouve que cette hypothèse est toujours vérifiée dans un typage principal : deux types de fermetures se terminant par la même variable d'expansion sont toujours égaux. Voici une explication intuitive de cette propriété. Les types de fermetures sont toujours créés avec des variables d'expansion différentes. La seule opération qui mène à partager une variable d'expansion entre plusieurs types de fermeture est l'identification de deux types fonctionnels $\tau_1 \multimap\langle\pi\rangle\!\rightarrowtail \tau_2$ et $\tau_1' \multimap\langle\pi'\rangle\!\rightarrowtail \tau_2'$. Mais cette opération va identifier π et π' tout entiers : il vont certes partager la même variable d'expansion, mais elle va être précédée du même ensemble de schémas dans les deux types de fermetures.

J'emploie l'adjectif "homogène" pour décrire cette situation où deux types de fermetures différents ne partagent jamais la même variable d'expansion. (Dans la suite de cette partie, on va définir plus formellement cette notion d'homogénéité.) On va se restreindre à des problèmes d'unification entre types homogènes, et dont les substitutions solutions ne seront appliquées qu'à des types homogènes avec les deux types de départ.

3.4.2 Types homogènes

Une CLASSIFICATION, notée K, est une application finie des variables d'extension (les variables de types de fermetures) dans les ensembles de schémas de types. On va maintenant définir ce qu'est un type, un schéma de types, ou un type de fermetures HOMOGÈNE avec K. On note $:: K$ cette relation d'homogénéité. Tout d'abord, un type de fermeture $\sigma_1, \ldots, \sigma_n, u$ est homogène avec K si K associe précisément l'ensemble $\{\sigma_1, \ldots, \sigma_n\}$ à la variable d'extension u. De plus, les σ_i doivent eux-mêmes être homogènes avec K.

$$\frac{\{\sigma_1 \ldots \sigma_n\} = K(u) \qquad \sigma_1 :: K \quad \ldots \quad \sigma_n :: K}{\sigma_1, \ldots, \sigma_n, u :: K}$$

Un type d'expression est homogène avec K si tous les types de fermetures qu'il contient le sont.

$$\iota :: K \qquad t :: K \qquad \frac{\tau_1 :: K \qquad \tau_2 :: K}{\tau_1 \times \tau_2 :: K} \qquad \frac{\tau_1 :: K \qquad \pi :: K \qquad \tau_2 :: K}{\tau_1 \multimap\langle\pi\rangle\!\rightarrowtail \tau_2 :: K}$$

$$\frac{\tau :: K}{\tau\ \texttt{ref} :: K} \qquad \frac{\tau :: K}{\tau\ \texttt{chan} :: K} \qquad \frac{\tau :: K}{\tau\ \texttt{cont} :: K}$$

Finalement, un schéma est homogène avec K si on peut trouver une extension K' de K à toutes les variables d'expansion universellement quantifiées dans le schéma telle que le type à l'intérieur du schéma est homogène avec K'.

$$\frac{\{u_1 \ldots u_m\} = \{\alpha_1, \ldots, \alpha_n\} \cap \texttt{VarTypClos} \qquad \tau :: K + u_1 \mapsto \Sigma_1 + \cdots + u_m \mapsto \Sigma_m}{(\forall \alpha_1 \ldots \alpha_n . \tau) :: K}$$

On étend immédiatement la relation $:: K$ aux environnements de typage en prenant $E :: K$ ssi $E(x) :: K$ pour tout $x \in \mathrm{Dom}(E)$.

Remarque. Si $\tau :: K$, alors le domaine de K contient toutes les variables d'expansion libres dans τ. Cette propriété vaut encore avec un schéma de type ou un type de fermeture à la place du type τ. □

Soit φ une substitution. On dit que φ est HOMOGÈNE DE K DANS K', et on note $\varphi :: K \Rightarrow K'$, si pour tout type τ tel que $\tau :: K$, on a $\varphi(\tau) :: K'$.

Remarque. Si $\varphi :: K \Rightarrow K'$ et $\varphi' :: K' \Rightarrow K''$, on a immédiatement $\varphi' \circ \varphi :: K \Rightarrow K''$. □

3.4.3 Unification

Dans cette partie, on donne un algorithme d'unification entre types K-homogènes, et on montre qu'il calcule un unificateur principal de ces types. Par la suite, on note Q un ensemble d'équations entre types d'expressions ($\tau_1 = \tau_2$) et entre types de fermetures ($\pi_1 = \pi_2$). Q ne contient donc pas d'équations "absurdes" de la forme $\tau = \pi$. (On fait donc de l'unification dans une algèbre de termes à deux sortes, la sorte des types d'expressions et la sorte des types de fermeture.)

Algorithme 3.1 Soit K une classification. Soit Q un ensemble d'équations entre types d'expressions et entre types de fermetures, tel que tous les types apparaissant dans Q sont K-homogènes. On définit une substitution $\mathtt{mgu}(Q)$ comme suit :

Si $Q = \emptyset$:
  $\mathtt{mgu}(Q) = [\,]$
Si $Q = \{\pi = \pi'\} \cup Q'$:
  écrivons $\pi = \sigma_1, \ldots, \sigma_n, u$ et $\pi' = \sigma'_1, \ldots, \sigma'_m, u'$
  si $u = u'$, alors $\mathtt{mgu}(Q) = \mathtt{mgu}(Q')$
  si $u \in \mathcal{L}(\pi')$ ou $u' \in \mathcal{L}(\pi)$, alors $\mathtt{mgu}(Q)$ n'est pas défini
  sinon on prend $\mathtt{mgu}(Q) = \mathtt{mgu}(\varphi(Q')) \circ \varphi$
  avec $\varphi = [u \mapsto (\sigma'_1, \ldots, \sigma'_m, u),\ u' \mapsto (\sigma_1, \ldots, \sigma_n, u)]$
Si $Q = \{t_1 = t_2\} \cup Q'$ et $t_1 = t_2$:
  $\mathtt{mgu}(Q) = \mathtt{mgu}(Q')$
Si $Q = \{t = \tau\} \cup Q'$ ou $Q = \{\tau = t\} \cup Q'$:
  si $t \in \mathcal{L}(\tau)$ alors $\mathtt{mgu}(Q)$ n'est pas défini
  sinon $\mathtt{mgu}(Q) = \mathtt{mgu}(\varphi(Q')) \circ \varphi$ avec $\varphi = [t \mapsto \tau]$
Si $Q = \{\iota_1 = \iota_2\} \cup Q'$ et $\iota_1 = \iota_2$:
  $\mathtt{mgu}(Q) = \mathtt{mgu}(Q')$
Si $Q = \{\tau_1 \multimap\!\langle\pi\rangle\!\rightarrowtail \tau_2 = \tau'_1 \multimap\!\langle\pi'\rangle\!\rightarrowtail \tau'_2\} \cup Q'$:
  $\mathtt{mgu}(Q) = \mathtt{mgu}(\{\tau_1 = \tau'_1,\ \pi = \pi',\ \tau_2 = \tau'_2\} \cup Q')$
Si $Q = \{\tau_1 \times \tau_2 = \tau'_1 \times \tau'_2\} \cup Q'$:
  $\mathtt{mgu}(Q) = \mathtt{mgu}(\{\tau_1 = \tau'_1,\ \tau_2 = \tau'_2\} \cup Q')$
Si $Q = \{\tau\ \mathtt{ref} = \tau'\ \mathtt{ref}\} \cup Q'$
ou $Q = \{\tau\ \mathtt{chan} = \tau'\ \mathtt{chan}\} \cup Q'$
ou $Q = \{\tau\ \mathtt{cont} = \tau'\ \mathtt{cont}\} \cup Q'$:

$$\texttt{mgu}(Q) = \texttt{mgu}(\{\tau = \tau'\} \cup Q')$$

Dans tous les autres cas, la substitution $\texttt{mgu}(Q)$ n'est pas définie.

Remarque. L'algorithme termine toujours, puisque à chaque étape la somme des hauteurs h des types dans Q diminue strictement (avec $h(t) = 1$, $h(\tau_1 \times \tau_2) = 1 + \max(h(\tau_1), h(\tau_2))$, etc.) □

Remarque. La substitution $\texttt{mgu}(Q)$ n'introduit pas de nouvelles variables par-rapport à Q : toute variable non libre dans Q est hors de portée de $\texttt{mgu}(Q)$. □

Proposition 3.13 *Soit Q un ensemble d'équations K-homogènes. Si $\mu = \texttt{mgu}(Q)$ est défini, alors μ est un unificateur de Q, et de plus il existe K' tel que $\mu :: K \Rightarrow K'$.*

Démonstration : on procède par récurrence sur le déroulement de l'algorithme. Mis à part le cas $\pi = \pi'$, tous les autres cas se prouvent comme pour l'algorithme classique de Robinson [84]. Les manipulations de classifications sont triviales dans ces cas-là : puisqu'on n'instancie pas de variables d'extension, les appels récursifs de `mgu` se font sur des systèmes eux aussi K-homogènes, et on peut prendre le même K' que celui fourni par l'hypothèse de récurrence. Je me contente donc de montrer le seul cas nouveau.

• **Cas $Q = \{\pi = \pi'\} \cup Q'$.** On écrit $\pi = \sigma_1, \ldots, \sigma_n, u$ et $\pi' = \sigma'_1, \ldots, \sigma'_m, u'$. Par hypothèse de K-homogénéité, on a $\{\sigma_1, \ldots, \sigma_n\} = K(u)$ et $\{\sigma'_1, \ldots, \sigma'_m\} = K(u')$.

Si $u = u'$, on a donc $\{\sigma_1, \ldots, \sigma_n\} = \{\sigma'_1, \ldots, \sigma'_m\}$, d'où $\pi = \pi'$ par application des axiomes de commutativité et d'idempotence. Si, comme nous le dit l'hypothèse de récurrence, $\texttt{mgu}(Q')$ est un unificateur de Q', alors c'est aussi un unificateur de Q tout entier.

Si $u \neq u'$, on sait que $u \notin \mathcal{L}(\pi')$ et $u' \notin \mathcal{L}(\pi)$. La substitution φ définie dans l'algorithme vérifie :

$$\begin{aligned} \varphi(\pi) &= \varphi(\sigma_1), \ldots, \varphi(\sigma_n), \varphi(u) \\ &= \varphi(\sigma_1), \ldots, \varphi(\sigma_n), \sigma'_1, \ldots, \sigma'_m, u \\ &= \sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m, u \end{aligned}$$

On a en effet $\varphi(\sigma_i) = \sigma_i$ pour tout i, puisque ni u ni u' ne sont libres dans σ_i. Si u' était libre dans σ_i, cela contredirait l'hypothèse $u' \notin \mathcal{L}(\pi)$. Si u était libre dans σ_i, cela contredirait l'hypothèse de K-homogénéité : u devrait apparaître dans σ_i précédée des mêmes schémas $\sigma_1, \ldots, \sigma_n$, ce qui n'est pas possible car π est de taille finie. Par symétrie, on a aussi

$$\varphi(\pi') = \sigma'_1, \ldots, \sigma'_m, \sigma_1, \ldots, \sigma_n, u.$$

La substitution φ est donc un unificateur de π et de π'. On définit

$$K_1 = \varphi(K) + u \mapsto \{\sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m\}.$$

On a $\varphi :: K \Rightarrow K_1$. En effet, pour tout type de fermeture $\pi :: K$, de trois choses l'une. Ou bien π se termine par u, et alors $\pi = \sigma_1, \ldots, \sigma_n, u$ par K-homogénéité, donc $\varphi(\pi) = \sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m, u$ est bien K_1-homogène. Ou bien π se termine par u', et alors $\pi = \sigma'_1, \ldots, \sigma'_m, u'$, et on conclut de

même. Ou bien π se termine par une variable d'expansion v qui n'est ni u ni u'. Donc $K_1(v) = \varphi(K(v))$, et $\varphi(\pi) :: \varphi(K)$ entraîne $\varphi(\pi) :: K_1$.

Il s'ensuit que $\varphi(Q')$ est K_1-homogène. Appliquant l'hypothèse de récurrence, il vient que $\texttt{mgu}(\varphi(Q'))$ est un unificateur de $\varphi(Q')$, et qu'il existe K' tel que $\texttt{mgu}(\varphi(Q')) :: K_1 \Rightarrow K'$. On en déduit que $\texttt{mgu}(\varphi(Q')) \circ \varphi$ est un unificateur de Q, et que $\texttt{mgu}(\varphi(Q')) \circ \varphi :: K \Rightarrow K'$. C'est le résultat désiré. □

On va maintenant montrer que $\texttt{mgu}(Q)$ est un unificateur principal de Q lorsqu'on ne considère que des types K-homogènes. Pour formaliser cette dernière notion, on dit que deux substitutions φ_1 et φ_2 sont K-ÉGALES, et on note $\varphi_1 \overset{K}{=} \varphi_2$, si $\varphi_1(\tau) = \varphi_2(\tau)$ pour tout type τ qui est K-homogène, et de même pour tous schémas σ et tous types de fermetures π à la place de τ.

Proposition 3.14 *Soit Q un ensemble d'équations K-homogènes. S'il existe une substitution ψ qui est un unificateur de Q, alors $\mu = \texttt{mgu}(Q)$ est défini, et il existe une substitution θ telle que $\psi \overset{K}{=} \theta \circ \mu$.*

Démonstration : on procède par récurrence sur la somme des tailles des types dans Q. Mis à part le cas $\pi = \pi'$, tous les autres cas se prouvent comme pour l'algorithme de Robinson.

• **Cas $Q = \{\pi = \pi'\} \cup Q'$.** On écrit $\pi = \sigma_1, \ldots, \sigma_n, u$ et $\pi' = \sigma'_1, \ldots, \sigma'_m, u'$. Si $u = u'$, on a forcément $\pi = \pi'$ par K-homogénéité, et le résultat est immédiat par l'hypothèse de récurrence. On suppose donc $u \neq u'$ dans ce qui suit. Montrons que $u \notin \mathcal{L}(\pi')$. Si tel était le cas, on aurait, par K-homogénéité, π tout entier qui apparaît comme sous-terme propre de π' ; alors $\psi(\pi)$ est sous-terme propre de $\psi(\pi')$, ce qui contredit $\psi(\pi) = \psi(\pi')$. Symétriquement, $u' \notin \mathcal{L}(\pi)$. Considérons la substitution φ de l'algorithme :

$$\varphi = [u \mapsto (\sigma'_1, \ldots, \sigma'_m, u),\ u' \mapsto (\sigma_1, \ldots, \sigma_n, u)]$$

On va montrer que $\psi \overset{K}{=} \psi \circ \varphi$. Soit $\pi_1 :: K$ un type de fermeture. On montre $\psi(\pi_1) = \psi(\varphi(\pi_1))$ par récurrence sur π_1. Si π_1 se termine par u, alors $\pi_1 = \pi$. Or,

$$\psi(\varphi(\pi)) = \psi(\sigma'_1, \ldots, \sigma'_m, \sigma_1, \ldots, \sigma_n, u) = \psi(\sigma'_1), \ldots, \psi(\sigma'_m), \psi(\pi) = \psi(\pi).$$

En effet, comme $\psi(\pi) = \psi(\pi')$, le type de fermeture $\psi(\pi)$ contient au moins $\psi(\sigma'_1), \ldots, \psi(\sigma'_m)$. Si π_1 se termine par u', alors $\pi_1 = \pi'$, et on a de même $\psi(\varphi(\pi')) = \psi(\pi')$. Dans tous les autres cas, π_1 se termine par une variable d'expansion v qui n'est ni u ni u'. Donc, $\varphi(v) = v$. De plus, par l'hypothèse de récurrence, les schémas qui précèdent v, étant K-homogènes, ont la même image par ψ et par $\psi \circ \varphi$. D'où $\psi(\pi_1) = \psi(\varphi(\pi_1))$ pour tout $\pi_1 :: K$.

D'autre part, comme $\psi \overset{K}{=} \psi \circ \varphi$, la substitution ψ est un unificateur de $\varphi(Q')$. De plus, $\varphi :: K \to K_1$, où la classification K_1 est construite comme dans la preuve de la proposition 3.13. Donc, $\varphi(Q')$ est K_1-homogène. Appliquant l'hypothèse de récurrence, il vient que $\texttt{mgu}(\varphi(Q'))$ est défini, et que $\psi \overset{K_1}{=} \theta \circ \texttt{mgu}(\varphi(Q'))$ pour une certaine substitution θ. En conclusion, $\texttt{mgu}(Q) = \texttt{mgu}(\varphi(Q')) \circ \varphi$ est bien défini, et

$$\psi \overset{K}{=} \psi \circ \varphi \overset{K}{=} \psi \circ \texttt{mgu}(\varphi(Q')) \circ \varphi = \theta \circ \texttt{mgu}(Q).$$

C'est le résultat recherché. □

3.4.4 L'algorithme d'inférence

Ayant défini une notion satisfaisante d'unificateur principal, il est facile d'adapter l'algorithme de Damas-Milner à l'inférence en présence de types de fermetures. L'algorithme prend en entrée une expression a, un environnement de typage E et un ensemble infini de "nouvelles" variables V ; il retourne un type τ (le type le plus général pour a), une substitution φ (représentant les instanciations qu'on a dû effectuer dans E), et un sous-ensemble V' de V (les "nouvelles" variables qu'on n'a pas utilisées).

On note $\texttt{Inst}(\sigma, V)$ une instance triviale du schéma σ. C'est-à-dire, supposant $\sigma = \forall\alpha_1 \dots \alpha_n.\,\tau$, on choisit n variables $\beta_1 \dots \beta_n$ dans V, avec β_i de la même sorte que α_i pour tout i, et on prend

$$\texttt{Inst}(\sigma, V) = ([\alpha_1 \mapsto \beta_1, \dots, \alpha_1 \mapsto \beta_1](\tau),\ V \setminus \{\beta_1 \dots \beta_n\}).$$

$\texttt{Inst}(\sigma, V)$ est défini à un renommage près des variables de V en variables de V.

Algorithme 3.2 $\texttt{Infer}(E, a, V)$ est le triplet (τ, φ, V') défini par :

Si a est x et $x \in \mathrm{Dom}(E)$:
  $(\tau, V') = \texttt{Inst}(E(x), V)$ et $\varphi = [\,]$
Si a est cst :
  $(\tau, V') = \texttt{Inst}(\texttt{TypCst}(cst), V)$ et $\varphi = [\,]$
Si a est $(f\ \texttt{where}\ f(x) = a_1)$:
  soient t et t' deux variables de types et u une variable d'extension, prises dans V
  soient $\{x_1, \dots, x_n\}$ les identificateurs libres dans $(f\ \texttt{where}\ f(x) = a_1)$
  soit $\pi = E(x_1), \dots, E(x_n), u$
  soit $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1,\ E + f \mapsto (t \multimap\langle\pi\rangle\rightarrow t') + x \mapsto t,\ V \setminus \{t, t', u\})$
  soit $\mu = \texttt{mgu}(\varphi_1(t'), \tau_1)$
  alors $\tau = \mu(\varphi_1(t \multimap\langle\pi\rangle\rightarrow t'))$ et $\varphi = \mu \circ \varphi_1$ et $V' = V_1$
Si a est $a_1(a_2)$:
  soit $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
  soit $(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), V_1)$
  soient $t \in V_2$ une variable de type et $u \in V_2$ une variable d'extension
  soit $\mu = \texttt{mgu}(\varphi_2(\tau_1), \tau_2 \multimap\langle u\rangle\rightarrow t)$
  alors $\tau = \mu(t)$ et $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ et $V' = V_2 \setminus \{t, u\}$
Si a est $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$:
  soit $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
  soit $(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E) + x \mapsto \texttt{Gen}(\tau_1, \varphi_1(E)), V_1)$
  alors $\tau = \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V = V_2$
Si a est (a_1, a_2) :
  soit $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
  soit $(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), V_1)$
  alors $\tau = \tau_1 \times \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V' = V_2$
Si a est $op(a_1)$:
  soit $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
  soit $(\tau_2, V_2) = \texttt{Inst}(\texttt{TypOp}(op), V_1)$
  soient $t \in V_2$ une variable de type et $u \in V_2$ une variable d'extension

soit $\mu = \mathtt{mgu}(\tau_1 \multimap\!\langle u\rangle\!\rightarrowtail t, \tau_2)$
alors $\tau = \mu(t)$ et $\varphi = \mu \circ \varphi_1$ et $V' = V_2 \setminus \{t, u\}$

On convient que $\mathtt{Infer}(a, E, V)$ n'est pas défini si, en cours de calcul, aucun cas ne s'applique ; en particulier, si on tente d'unifier deux types non unifiables. $\mathtt{Infer}(a, E, V)$ est défini à un renommage près des variables de V en variables de V.

Proposition 3.15 (Correction de l'algorithme d'inférence) *Soient a une expression, E un environnement de typage et V un ensemble infini de variables de types. Si $(\tau, \varphi, V') = \mathtt{Infer}(a, E, V)$ est défini, alors on peut dériver $\varphi(E) \vdash a : \tau$.*

Démonstration : la preuve décalque exactement celle de la proposition 1.8, et repose essentiellement sur la stabilité du jugement de typage par substitution (proposition 3.2). □

Proposition 3.16 (Complétude de l'algorithme d'inférence) *Soient K une classification, a une expression, $E :: K$ un environnement de typage, et V un ensemble de variables contenant une infinité de variable de types et une infinité de variables de types de fermetures, et telles que $V \cap \mathcal{L}(E) = \emptyset$. S'il existe un type τ' et une substitution φ' telle que $\varphi'(E) \vdash a : \tau'$, alors $(\tau, \varphi, V') = \mathtt{Infer}(a, E, V)$ est bien défini, et il existe une substitution ψ telle que*

$$\tau' = \psi(\tau) \quad et \quad \varphi' \overset{K}{=} \psi \circ \varphi \text{ hors de } V.$$

(C'est-à-dire, $\varphi'(\tau) = \psi(\varphi(\tau))$ pour tout type $\tau :: K$ tel que $\mathcal{L}(\tau) \cap V = \emptyset$.)

Démonstration : remarquons tout d'abord que si $(\tau, \varphi, V') = \mathtt{Infer}(a, E, V)$ est défini, alors il existe K' tel que $\tau :: K'$ et $\varphi :: K \Rightarrow K'$. De plus, $V' \subseteq V$, et les variables de V' ne sont pas libres dans τ et sont hors de portée de φ. On le montre par une récurrence facile sur le déroulement de l'algorithme, en utilisant la proposition 3.13 et le fait que l'unificateur $\mathtt{mgu}(\tau_1, \tau_2)$ n'introduit pas de nouvelles variables. En conséquence de ces remarques, $\varphi(E) :: K'$ et $V' \cap \mathcal{L}(\varphi(E)) = \emptyset$.

La preuve de la proposition est par récurrence sur la dérivation de $\varphi'(E) \vdash a : \tau'$, et par cas sur a. La preuve procède exactement comme celle de la proposition 1.9, avec des manipulations de classifications K en plus. Je montre un cas, pour illustrer l'utilisation de l'hypothèse de K-homogénéité.

• **Cas $a = a_1(a_2)$.** La dérivation initiale est de la forme

$$\frac{\varphi'(E) \vdash a_1 : \tau'' \multimap\!\langle\pi'\rangle\!\rightarrowtail \tau' \qquad \varphi'(E) \vdash a_2 : \tau''}{\varphi'(E) \vdash a_1(a_2) : \tau'}$$

On applique l'hypothèse de récurrence à a_1, $E :: K$, V, $\tau'' \multimap\!\langle u'\rangle\!\rightarrowtail \tau'$ et φ'. Il vient

$$(\tau_1, \varphi_1, V_1) = \mathtt{Infer}(a_1, E, V) \quad \text{et} \quad \tau'' \multimap\!\langle\pi'\rangle\!\rightarrowtail \tau' = \psi_1(\tau_1) \quad \text{et} \quad \varphi' \overset{K}{=} \psi_1 \circ \varphi_1 \text{ hors de } V \quad \text{et}$$

$$\tau_1 :: K_1 \quad \text{et} \quad \varphi_1 :: K \Rightarrow K_1.$$

En particulier, $\varphi'(E) = \psi_1(\varphi_1(E))$ et $\varphi_1(E) :: K_1$. On applique l'hypothèse de récurrence à a_2, $\varphi_1(E) :: K_1$, V_1, τ et ψ_1. On a bien $\mathcal{L}(\varphi_1(E)) \cap V_1 = \emptyset$ par la remarque du début de la preuve. Il vient :

$$(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), V_1) \quad \text{et} \quad \tau'' = \psi_2(\tau_2) \quad \text{et} \quad \psi_1 \overset{K_1}{=} \psi_2 \circ \varphi_2 \text{ hors de } V_1 \quad \text{et}$$

$$\tau_2 :: K_2 \quad \text{et} \quad \varphi_2 :: K_1 \Rightarrow K_2.$$

On a $\mathcal{L}(\tau_1) \cap V_1 = \emptyset$, d'où $\psi_1(\tau_1) = \psi_2(\varphi_2(\tau_1))$. Posons

$$\psi_3 = \psi_2 + t \mapsto \tau' + u \mapsto \pi' \qquad\qquad K_3 = K_2 + u \mapsto \Sigma' \text{ avec } (\Sigma', u') = \pi'.$$

Les variables t et u, choisies dans V_2, sont hors de portée de ψ_2, et donc ψ_3 prolonge ψ_2. De même, on peut supposer $u \notin \mathrm{Dom}(K_2)$, et donc K_3 prolonge K_2. On a donc :

$$\begin{array}{rcccccl} \psi_3(\varphi_2(\tau_1)) & = & \psi_2(\varphi_2(\tau_1)) & = & \psi_1(\tau_1) & = & \tau'' \to \tau' \\ \psi_3(\tau_2 \to \alpha) & = & \psi_2(\tau_2) \to \tau'' & = & \tau'' \to \tau' & & \end{array}$$

La substitution φ_3 est donc un unificateur de $\varphi_2(\tau_1)$ et $\tau_2 \dashv\langle u \rangle\rightarrow t$. De plus, ces deux types sont K_3-homogènes. L'unificateur principal de ces deux types, μ, existe donc, et $\texttt{Infer}(a_1(a_2), E, V)$ est bien défini. De plus, on a $\mu :: K_3 \Rightarrow K_4$ et $\psi_3 = \psi_4 \circ \mu$ pour une certaine substitution ψ_4 et une certaine classification K_4. On montre maintenant que $\psi = \psi_4$ et $K' = K_4$ conviennent. Avec les notations de l'algorithme, on a bien

$$\psi(\tau) = \psi_4(\mu(\alpha)))) = \psi_3(\alpha) = \tau'.$$

De plus, pour tout $\tau :: K$ tel que $\mathcal{L}(\tau) \cap V = \emptyset$ (et donc a fortiori $\mathcal{L}(\tau) \cap V_1 = \emptyset$, $\beta \notin V_2$, $\beta \neq \alpha$) :

$$\begin{array}{rcll} \psi(\varphi(\tau)) & = & \psi_4(\mu(\varphi_2(\varphi_1(\tau)))) & \text{par définition de } \varphi \\ & = & \psi_3(\varphi_2(\varphi_1(\tau))) & \text{par définition de } \psi_4 \\ & = & \psi_2(\varphi_2(\varphi_1(\tau))) & \text{parce que } t \notin \mathcal{L}(\tau) \text{ et } t \text{ hors de portée de de } \varphi_1 \text{ et de } \varphi_2 \\ & = & \psi_1(\varphi_1(\tau)) & \text{parce que } \varphi_1(\tau) :: K_1 \text{ et } \mathcal{L}(\tau) \cap V_1 = \emptyset \\ & = & \varphi'(\tau) & \text{parce que } \tau :: K \text{ et } \mathcal{L}(\tau) \cap V = \emptyset. \end{array}$$

Enfin, on a immédiatement $\varphi :: K \Rightarrow K_4$. Ceci achève d'établir le résultat annoncé. □

Chapitre 4

Typage fin des fermetures

Dans ce chapitre, on introduit et on étudie une variante du système de types présenté au chapitre 3, qui repose sur les mêmes idées de variables dangereuses et de typage des fermetures, mais qui effectue le typage des fermetures d'une manière plus fine. Le typage des fermetures, tel qu'il est réalisé dans le système du chapitre 3, se révèle parfois trop faible, comme on va le montrer tout de suite ; le but de ce deuxième système est de remédier à ces faiblesses.

4.1 Non-conservativité du système initial

Le système de types proposé au chapitre 3, bien qu'assurant la sûreté de l'exécution, n'est pas entièrement satisfaisant. Le problème est que ce système rejette comme mal typés certains programmes purement applicatifs (c'est-à-dire, n'utilisant ni références, ni canaux, ni continuations) qui sont bien typés en ML. J'appelle "conservativité" cette propriété qu'un système de types pour des extensions algorithmiques de ML accepte de typer tous les programmes bien typés en ML "pur". La conservativité est la marque que le système de types est bien une extension de celui de ML.

La non-conservativité ne provient pas de la restriction de la généralisation aux variables non dangereuses : aucune variable n'est dangereuse dans un programme purement applicatif. Le problème est dans le typage des fermetures : même dans un programme purement applicatif, le typage des fermetures mène à des types fonctionnels plus riches, donc éventuellement plus sélectifs, d'une part, et d'autre part pouvant contenir davantage de variables libres, ce qui peut empêcher la généralisation de certaines variables dans d'autres types.

4.1.1 Types de fermetures récursifs

Le typage des fermetures conduit à attribuer des types différents à des fonctions qui ont le même type en ML. Par exemple, deux fonctions des entiers dans les entiers peuvent avoir deux types différents $\texttt{int} \multimap\langle\sigma_1, u\rangle\rightarrowtail \texttt{int}$ et $\texttt{int} \multimap\langle\sigma_2, v\rangle\rightarrowtail \texttt{int}$, alors qu'en ML elles ont le même type $\texttt{int} \to \texttt{int}$. La plupart du temps, ceci n'est pas gênant : on peut "presque toujours" identifier deux types de fermetures quelconques, en instanciant convenablement leurs variables d'extension. Dans l'exemple précédent, si u et v n'apparaissent pas dans σ_1 ni dans σ_2, on peut identifier les deux types en

remplaçant u par σ_2, w et v par σ_1, w. Malheureusement, l'identification n'est pas toujours possible dans le cas où les variables d'extension des types de fermetures apparaissent aussi à l'intérieur des types de fermetures, dans un des schémas. Par exemple, il n'y a pas d'instance commune aux deux types

$$\tau_1 \multimap\langle u\rangle\rightarrow \tau_2 \quad \text{et} \quad \tau_1 \multimap\langle \tau_1 \multimap\langle u\rangle\rightarrow \tau_2, v\rangle\rightarrow \tau_2.$$

Pourtant, si `f` a pour type le type de gauche, le type de droite est le type de la forme eta-expansée de `f`, c'est-à-dire λ`x.f(x)`. Et puisque ces types sont incompatibles, cela veut dire que la phrase suivante est mal typée :

$$\lambda\texttt{f.if ... then f else } \lambda\texttt{x.f(x)}.$$

Elle est pourtant parfaitement correcte — et bien typée en ML.

On peut contourner ce problème en autorisant les types de fermeture récursifs : des types de fermetures de la forme $\mu u.\,\pi$, représentant en fait le type infini vérifiant $\pi = u$. Dans l'exemple de l'eta-expansion, les deux types auparavant incompatibles admettent maintenant comme instance commune

$$\tau_1 \multimap\langle \mu u.\,\tau_1 \multimap\langle u\rangle\rightarrow \tau_2, v\rangle\rightarrow \tau_2,$$

obtenue en substituant u par $\mu u.\,\tau_1 \multimap\langle u\rangle\rightarrow \tau_2, v$ dans le type de droite, et par la forme équivalente $\tau_1 \multimap\langle \mu u.\,\tau_1 \multimap\langle u\rangle\rightarrow \tau_2, v\rangle\rightarrow \tau_2, v$ dans le type de gauche. Cette instance commune représente en fait le type infini suivant :

$$\tau_1 \multimap\langle \tau_1 \multimap\langle \tau_1 \multimap\langle \tau_1 \multimap\langle \ldots, v\rangle\rightarrow \tau_2,\ v\rangle\rightarrow \tau_2,\ v\rangle\rightarrow \tau_2,\ v\rangle\rightarrow \tau_2.$$

Plus généralement, on peut montrer que, si on autorise les types de fermetures récursifs, deux types de fermetures quelconques admettent toujours une instance commune. Et donc, deux fonctions ont des types compatibles si et seulement si les types de leur argument sont compatibles, ainsi que les types de leur résultat, exactement comme dans le système de types de ML. (Dans la suite de ce chapitre, on va donner une présentation différente des types de fermetures, qui n'utilise pas des types récursifs de la forme $\mu u.\,\pi$. Cette autre présentation possède la même propriété que deux types de fermetures quelconques ont toujours une instance commune.)

4.1.2 Capture de variables dans les types de fermetures

Même si on s'arrange pour que deux types de fermetures soient toujours compatibles, il reste encore des programmes ML "purs" qui ne sont pas bien typés lorsqu'on effectue le typage des fermetures. Il y a en effet une autre raison pour laquelle le typage des fermetures peut conduire à attribuer des schémas de types moins généraux qu'en ML : des variables de types peuvent apparaître libres dans un type fonctionnel avec type de fermetures, alors qu'elles ne sont pas libres dans le type ML correspondant. Par exemple, t est libre dans $\texttt{int} \multimap\langle t\ \texttt{list}, u\rangle\rightarrow \texttt{int}$, alors qu'elle n'est pas libre dans $\texttt{int} \rightarrow \texttt{int}$. Si le type $\texttt{int} \multimap\langle t\ \texttt{list}, u\rangle\rightarrow \texttt{int}$ fait partie de l'environnement de typage courant, cela veut dire que la variable t n'est pas généralisable, alors qu'elle serait généralisable dans le système de types de ML. Voici un exemple où ce phénomène se produit :

$$\lambda\texttt{f. let id} = \lambda\texttt{y. either(f)(}\lambda\texttt{z.y;z);\ y in id(id)}$$

où la fonction `either` est définie comme :

$$\texttt{let either} = \lambda\texttt{x}.\,\lambda\texttt{y}.\ \texttt{if ... then x else y}$$

et a pour but de forcer ses deux arguments à avoir le même type. Tentons de typer cette phrase, en prenant comme hypothèse $\texttt{y} : t_1$. La fonction $\lambda\texttt{z}.\,\texttt{y};\texttt{z}$ a le type $t_2 \multimap\langle t_1, u\rangle\rightarrowtail t_2$. Par l'effet de `either`, le paramètre `f` a le même type. Le membre gauche du `let`, $\lambda\texttt{y}\ldots$, a un type de la forme $t_1 \multimap\langle \pi'\rangle\rightarrowtail t_1$. Au moment où l'on généralise ce dernier type, l'environnement de typage est

$$\texttt{f} : t_2 \multimap\langle t_1, u\rangle\rightarrowtail t_2.$$

La variable t_1 est libre dans cet environnement, et donc non généralisable dans $t_1 \multimap\langle \pi'\rangle\rightarrowtail t_1$. Par conséquent, `id` reste monomorphe, et l'application `id(id)` n'est pas bien typée. La même phrase est bien typée en ML, car, sans typage des fermetures, t_1 n'apparaît pas dans le type de `f`.

Ce phénomène de capture de variables par l'intermédiaire des types de fermetures est infiniment plus coriace à éliminer que les phénomènes d'incompatibilité entre types de fermetures. Une première manière de l'éviter est d'ignorer, pour le calcul des variables libres d'un type fonctionnel $\tau_1 \multimap\langle\pi\rangle\rightarrowtail \tau_2$, toutes les variables qui sont libres dans π, mais pas dans τ_1 ni dans τ_2. Une autre manière est, lorsqu'on donne le type $\tau_1 \multimap\langle\pi\rangle\rightarrowtail \tau_2$ à une fonction (f `where` $f(x) = a$), de ne pas mettre dans π le type d'un identificateur libre dans (f `where` $f(x) = a$) si ce type n'a pas de variables libres en commun avec le type τ_1 de l'argument ou le type τ_2 du résultat.

Ne risque-t'on pas de perdre alors la trace de certaines références polymorphes cachées dans des fermetures ? Par exemple, dans une valeur fonctionnelle de type $\alpha \multimap\langle\beta\ \texttt{ref}, u\rangle\rightarrowtail \alpha$ se trouve une référence de type β `ref`. Avec les deux manières d'éviter les captures de variables proposées plus haut, on va ignorer le fait que β est dangereuse dans ce type, et donc on va pouvoir généraliser β, attribuant ainsi un type polymorphe à une référence. Est-ce grave ? Non. Car on ne peut pas accéder à cette référence polymorphe. La fonction ne peut pas stocker (des morceaux de) son argument dans cette référence, puisque l'argument est de type α alors que la référence est de type β. La fonction ne peut pas renvoyer la référence dans son résultat, puisque le résultat est de type α alors que la référence est de type β. Enfin, la fonction ne peut pas stocker la référence dans un autre référence, accessible celle-ci, puisque cette autre référence devrait être de type β `ref ref` et accessible à partir de l'environnement, et donc β aurait été considérée dangereuse au moment de la généralisation.

L'idée qu'il y a derrière ce raisonnement est la suivante : si on peut typer une expression sous les hypothèse $x : \alpha$ et $y : \beta$ (deux variables de types distinctes), alors pendant l'exécution de l'expression, il n'y a pas de communication possible entre les valeurs de x et de y. Cette intuition est à la base de certaines adaptations de l'algorithme de Damas-Milner pour l'inférence statique de propriétés de partage entre données [5]. Je ne connais pas de formulation précise de ce résultat (comment caractériser la non-communication ?), ni à plus forte raison de preuve.

4.1.3 Enjeux de la conservativité

La propriété de conservativité est fortement souhaitable. Le système de ML est généralement considéré comme un système de typage satisfaisant pour un langage purement applicatif. Il est donc préférable de ne pas être plus restrictif que lui. Aussi, il est satisfaisant pour l'esprit que les mécanismes de typage mis en place pour contrôler les extensions “impures” du langage n'interviennent

pas quand on n'utilise pas ces extensions. De ce point de vue, le système présenté au chapitre 3 n'est pas satisfaisant.

D'un point de vue plus pragmatique, il faut noter que la non-conservativité de ce système ne se manifeste que sur des exemples artificiels et assez compliqués. En particulier, je n'ai pas encore rencontré de programme réaliste qui se heurte au problème de la capture de variables via les types de fermetures. Pour le vérifier, j'ai équipé Caml Light [49], mon compilateur ML, du système de types décrit au chapitre 3, et je lui ai fait avaler une dizaine de milliers de lignes de ML ; le phénomène de capture ne s'est pas manifesté. J'affirme donc que le système du chapitre 3 est conservatif pour tous les usages pratiques.

Ce genre d'affirmations ne possède cependant pas la rigueur mathématique qui sied à une thèse de doctorat d'Informatique Théorique de l'université Paris 7. Mes tests ne sont évidemment pas exhaustifs : je n'ai considéré que des programmes relativement simples, laissant de côté certaines tournures utilisant la fonctionnalité à haute dose qui peuvent apparaître par exemple dans des programmes extraits à partir de preuves [24], ou dans des codages astucieux de structures de données complexes [23].

J'ai donc passé beaucoup de temps à chercher un système de types avec variables dangereuses et types de fermetures qui possède la propriété de conservativité. Le reste de ce chapitre présente le résultat de cette recherche. Pour atteindre la conservativité et éviter les phénomènes de capture, un contrôle très fin de la généralisation est nécessaire — beaucoup plus fin que dans le système du chapitre 3. J'ai donc du m'écarter de la présentation assez classique du typage de ML adoptée aux chapitres 1 et 3, et adopter une présentation à base de graphes, plus inhabituelle.

4.2 Le système de types indirect

4.2.1 Présentation

Le système de types du présent chapitre se distingue du système du chapitre 3 par deux aspects. L'un concerne la représentation des types de fermetures ; l'autre, la représentation des types polymorphes.

4.2.1.1 Étiquettes plus contraintes au lieu de types de fermetures

L'idée de départ de ce système est d'ajouter un niveau d'indirection supplémentaire dans l'association type fonctionnel/type de fermetures. Au lieu d'annoter directement les types de fonctions par les ensembles extensibles de schémas de types qui constituent les types de fermetures, on se contente d'annoter les types de fonctions $\tau_1 \to \tau_2$ par une simple variable, qu'on appelle une ÉTIQUETTE et qu'on note u. Les types de fonctions sont donc de la forme $\tau_1 \multimap\!\langle u \rangle\!\!\to \tau_2$. En dehors des expressions de types, dans un environnement séparé, on associe à chaque étiquette u un ensemble de schémas de types : les types des valeurs contenues dans les fermetures étiquetées u. Cet environnement séparé se présente comme un ensemble de CONTRAINTES de la forme $\sigma_1 \lhd u_1, \ldots, \sigma_n \lhd u_n$. La contrainte $\sigma \lhd u$ doit se lire comme "toute fermeture étiquetée u peut contenir une valeur de type σ".

On va donc manipuler des couples (τ, C) d'une expression de type τ, avec les types fonctionnels annotés par des étiquettes, et d'un ensemble de contraintes C définissant le contenu des étiquettes.

On notera plutôt $\tau \,/\, C$ de tels couples, et on les appellera types contraints. Un type contraint joue à peu près le même rôle qu'une expression de types dans le système du chapitre 3 : au type contraint

$$\tau_1 \multimap\langle u\rangle\rightarrow \tau_2 \,/\, C$$

correspond, dans l'algèbre de types du chapitre 3, le type direct

$$\tau_1 \multimap\langle \sigma_1, \ldots, \sigma_n, u\rangle\rightarrow \tau_2$$

où $\sigma_1 \ldots \sigma_n$ sont les schémas σ associés à u dans C (c'est-à-dire, tels que la contrainte $\sigma \triangleleft u$ apparaît dans C).

La représentation indirecte se révèle plus puissante que la représentation directe sur plusieurs points. Premièrement, les types de fermetures peuvent naturellement être récursifs : dans une contrainte $\sigma \triangleleft u$, l'étiquette u peut très bien apparaître à nouveau dans σ. Par exemple, le type contraint

$$\tau_1 \multimap\langle v\rangle\rightarrow \tau_2 \,/\, \tau_1 \multimap\langle v\rangle\rightarrow \tau_2 \triangleleft v$$

représente très simplement le type infini dont on avait besoin dans l'exemple de la eta-expansion,

$$\tau_1 \multimap\langle \tau_1 \multimap\langle \tau_1 \multimap\langle \ldots, v\rangle\rightarrow \tau_2,\, v\rangle\rightarrow \tau_2,\, v\rangle\rightarrow \tau_2.$$

Deuxièmement, cette représentation assure syntaxiquement que les types de fermetures sont toujours homogènes, au sens de la partie 3.4.2. Les preuves sur l'algorithme d'inférence de types s'en trouvent simplifiées : il n'y a plus besoin de manipuler des sortes.

Contexte. Cette représentation m'a été proposée par Didier Rémy, début 1990. Elle semble inspirée par le traitement des hypothèses de sous-typage dans les systèmes d'inférence en présence de sous-types [65, 28]. Cette idée de représentation indirecte des types a, depuis la parution de [51], été reprise et appliquée aux problèmes d'inférence dans les systèmes d'effets [89, 99]. □

4.2.1.2 Types génériques au lieu de schémas de types

Deuxième différence entre le système qu'on va présenter et ceux qu'on a déjà vu : les schémas de types y sont représentés non pas en quantifiant universellement des variables en tête d'un type, mais en marquant spécialement les variables qu'on veut généraliser. On sépare donc les variables en deux classes, les variables génériques (notées avec un indice g) et les variables non génériques (notées avec un indice n). Les schémas de types sont représentés par des types génériques, encore notés σ : des types qui peuvent contenir des variables génériques. Les types simples sont représentés par des types non génériques, encore notés τ : des types dont toutes les variables sont non génériques. L'opération d'instanciation revient à substituer les variables génériques d'un type générique par des types non génériques, obtenant ainsi un type non générique : l'instance. L'opération de généralisation revient à renommer des variables non génériques en variables génériques.

Exemple. Le type simple $\alpha \times \texttt{int}$, dans l'ancienne notation, devient le type non générique $\tau = \alpha_n \times \texttt{int}$. Le schéma $\forall\beta.\, \alpha \times \beta$ devient le type générique $\sigma = \alpha_n \times \beta_g$. On a bien τ instance de σ, par substitution de β_g par $\texttt{int}$. □

Contexte. Cette représentation est employée, sous une forme légèrement différente, dans certaines implémentations de l'algorithme de Damas-Milner [11]. Elle permet d'avoir une seule représentation pour les types et pour les schémas, et de pouvoir déterminer localement si une variable est générique ou non. Dans sa thèse [77, chapitre 3], Didier Rémy donne une formulation du typage de ML en termes de types génériques/non génériques, et montre son équivalence avec la formulation habituelle en termes de types simples/schémas de types. □

L'intérêt de cette représentation des schémas est que les variables génériques ont une portée illimitée, alors que les variables quantifiées universellement le sont seulement dans la portée du $\forall$. Cette portée limitée de la quantification pose problème lorsque les types de fermetures sont représentés de manière indirecte. D'un côté, il se révèle important que les contraintes décrivant le contenu des étiquettes soient partagées entre tous les types. Ceci nécessite un jugement de typage de la forme $(E \vdash a : \tau) \,/\, C$, où les contraintes de C s'appliquent à la fois au type τ et aux schémas de types contenus dans E. D'un autre côté, certaines contraintes peuvent faire (moralement) partie d'un des schémas de E, et donc contenir des variables généralisées. Mais ces contraintes sont en-dehors de la portée des quantificateurs universels qu'on pourrait mettre dans E.

Le recours aux variables génériques lève cette difficulté : une même variable générique peut apparaître à la fois dans un type générique de E et dans l'ensemble de contraintes courant C ; lorsqu'on prend une instance de ce type générique, on substitue la variable générique par un type non générique aussi bien dans le type générique que dans l'ensemble de contraintes C.

Contexte. On peut quand même combiner quantification universelle et types indirects : il faut considérer des schémas de types de la forme $\forall\alpha_1 \ldots \alpha_n.(\tau \,/\, C)$, où C est un ensemble de contraintes locales sur les variables universellement quantifiées, qui s'additionne aux contraintes globales lorsqu'on prend une instance du schéma. C'est ainsi qu'est présentée une première version des travaux de cette Thèse [51]. Cependant, ces contraintes locales posent des problèmes techniques lorsqu'on cherche, comme dans le présent chapitre, à éviter les phénomènes de capture de variables via les types de fermeture. □

4.2.2 L'algèbre de types

4.2.2.1 Variables de types

On se donne quatre ensembles infinis de variables de types :

t_n	$\in$	`VarTypExpNongen`	variables non génériques de types d'expressions
t_g	$\in$	`VarTypExpGen`	variables génériques de types d'expressions
u_n	$\in$	`EtiqNongen`	étiquettes non génériques de types de fermetures
u_g	$\in$	`EtiqGen`	étiquettes génériques de types de fermetures

On nomme les réunions deux à deux de ces ensembles de variables comme suit :

$$
\begin{array}{rcl}
t & \in & \texttt{VarTypExp} = \texttt{VarTypExpNongen} \cup \texttt{VarTypExpGen} \\
 & & \quad \text{variables de types d'expressions, génériques ou non} \\
u & \in & \texttt{Etiq} = \texttt{EtiqNongen} \cup \texttt{EtiqGen} \\
 & & \quad \text{étiquettes, génériques ou non} \\
\alpha_n & \in & \texttt{VarTypNongen} = \texttt{VarTypExpNongen} \cup \texttt{EtiqNongen} \\
 & & \quad \text{variables non génériques de types, d'expressions ou de fermetures} \\
\alpha_g & \in & \texttt{VarTypGen} = \texttt{VarTypExpGen} \cup \texttt{EtiqGen} \\
 & & \quad \text{variables génériques de types, d'expressions ou de fermetures}
\end{array}
$$

4.2.2.2 Expressions de types

L'ensemble `TypGen` des TYPES GÉNÉRIQUES (notés σ) est défini par la grammaire suivante.

$$
\begin{array}{rcll}
\sigma & ::= & \iota & \text{type de base} \\
 & | & t & \text{variable (générique ou non)} \\
 & | & \sigma_1 \multimap\langle u\rangle\rightarrow \sigma_2 & \text{type fonctionnel (étiqueté)} \\
 & | & \sigma_1 \times \sigma_2 & \text{type produit} \\
 & | & \sigma\ \texttt{ref} & \text{type de référence} \\
 & | & \sigma\ \texttt{chan} & \text{type de canal} \\
 & | & \sigma\ \texttt{cont} & \text{type de continuation}
\end{array}
$$

L'ensemble `TypNongen` des TYPES NON GÉNÉRIQUES (notés τ) est constitué des types génériques qui ne contiennent aucune variable générique de types. Ce sous-ensemble est décrit par la grammaire suivante.

$$
\begin{array}{rcll}
\tau & ::= & \iota & \text{type de base} \\
 & | & t_n & \text{variable (non générique)} \\
 & | & \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2 & \text{type fonctionnel (étiqueté)} \\
 & | & \tau_1 \times \tau_2 & \text{type produit} \\
 & | & \tau\ \texttt{ref} & \text{type de référence} \\
 & | & \tau\ \texttt{chan} & \text{type de canal} \\
 & | & \tau\ \texttt{cont} & \text{type de continuation}
\end{array}
$$

Les CONTRAINTES sont des couples d'un type générique et d'une étiquette (générique ou non). On les note $\sigma \triangleleft u$ (lire : "σ est dans u"). Les ensembles de contraintes sont notés C.

$$C \quad ::= \quad \{\sigma_1 \triangleleft u_1, \ldots, \sigma_n \triangleleft u_n\} \quad \text{ensembles de contraintes}$$

4.2.2.3 Substitutions

Les substitutions sur cette algèbre de types sont des applications finies des variables de types d'expressions dans les types d'expressions, et des étiquettes dans les étiquettes :

$$\text{Substitutions :} \quad \varphi, \psi \quad ::= \quad [t \mapsto \sigma, \ldots, u \mapsto u', \ldots]$$

On aura souvent besoin de préciser le domaine ou l'image d'une substitution. A cet effet, on note $\varphi : V \Rightarrow T$ pour dire que φ est une substitution de domaine inclus dans l'ensemble de variables V,

et d'image incluse dans l'ensemble de types et d'étiquettes T. De même, on note $\varphi : V \Leftrightarrow V'$ pour dire que φ est un renommage de toutes les variables de V en des variables de V'. Un RENOMMAGE est une substitution injective dont l'image se compose uniquement de variables. Tout renommage $\varphi : V \Leftrightarrow V'$ admet un inverse, noté φ^{-1}.

Une substitution φ s'étend naturellement en un morphisme $\overline{\varphi}$ de types d'expressions et d'étiquettes de fermetures, de la manière suivante :

$$\begin{array}{rcl}
\overline{\varphi}(\alpha) & = & \varphi(\alpha) \text{ si } \alpha \in \mathrm{Dom}(\varphi) \\
\overline{\varphi}(\alpha) & = & \alpha \text{ si } \alpha \notin \mathrm{Dom}(\varphi) \\
\overline{\varphi}(\iota) & = & \iota \\
\overline{\varphi}(\sigma_1 \multimap\langle u\rangle\rightarrowtail \sigma_2) & = & \overline{\varphi}(\sigma_1) \multimap\langle \overline{\varphi}(u)\rangle\rightarrowtail \overline{\varphi}(\sigma_2) \\
\overline{\varphi}(\sigma_1 \times \sigma_2) & = & \overline{\varphi}(\sigma_1) \times \overline{\varphi}(\sigma_2) \\
\overline{\varphi}(\sigma\ \texttt{ref}) & = & \overline{\varphi}(\sigma)\ \texttt{ref} \\
\overline{\varphi}(\sigma\ \texttt{chan}) & = & \overline{\varphi}(\sigma)\ \texttt{chan} \\
\overline{\varphi}(\sigma\ \texttt{cont}) & = & \overline{\varphi}(\sigma)\ \texttt{cont}
\end{array}$$

Pour les ensembles de contraintes, on prend naturellement :

$$\overline{\varphi}(C) = \{(\overline{\varphi}(\sigma) \lhd \overline{\varphi}(u)) \mid (\sigma \lhd u) \in C\}.$$

À partir de maintenant, on confond φ et son extension $\overline{\varphi}$, et on note φ pour les deux.

4.2.2.4 Variables libres, variables dangereuses

Soit σ un type générique. On définit les variables directement libres $\mathcal{L}(\sigma)$ et les variables directement dangereuses $\mathcal{D}(\sigma)$ dans le type σ comme suit.

$$\begin{array}{rcl}
\mathcal{L}(\iota) & = & \emptyset \\
\mathcal{L}(t) & = & \{t\} \\
\mathcal{L}(\sigma_1 \multimap\langle u\rangle\rightarrowtail \sigma_2) & = & \mathcal{L}(\sigma_1) \cup \mathcal{L}(u) \cup \mathcal{L}(\sigma_2) \\
\mathcal{L}(\sigma_1 \times \sigma_2) & = & \mathcal{L}(\sigma_1) \cup \mathcal{L}(\sigma_2) \\
\mathcal{L}(\sigma\ \texttt{ref}) & = & \mathcal{L}(\sigma) \\
\mathcal{L}(\sigma\ \texttt{chan}) & = & \mathcal{L}(\sigma) \\
\mathcal{L}(\sigma\ \texttt{cont}) & = & \mathcal{L}(\sigma) \\
\mathcal{L}(u) & = & \{u\}
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{D}(\iota) & = & \emptyset \\
\mathcal{D}(t) & = & \emptyset \\
\mathcal{D}(\sigma_1 \multimap\langle u\rangle\rightarrowtail \sigma_2) & = & \mathcal{D}(u) \\
\mathcal{D}(\sigma_1 \times \sigma_2) & = & \mathcal{D}(\sigma_1) \cup \mathcal{D}(\sigma_2) \\
\mathcal{D}(\sigma\ \texttt{ref}) & = & \mathcal{L}(\sigma) \\
\mathcal{D}(\sigma\ \texttt{chan}) & = & \mathcal{L}(\sigma) \\
\mathcal{D}(\sigma\ \texttt{cont}) & = & \mathcal{L}(\sigma) \\
\mathcal{D}(u) & = & \emptyset
\end{array}$$

Voici comment évoluent variables libres et dangereuses lorsqu'on applique une substitution.

Proposition 4.1 *Soient σ un type générique et φ une substitution. On a :*

$$\begin{array}{rcl}
\mathcal{L}(\varphi(\sigma)) & = & \displaystyle\bigcup_{\alpha\in\mathcal{L}(\sigma)} \mathcal{L}(\varphi(\alpha)) \\
\mathcal{D}(\varphi(\sigma)) & \supseteq & \displaystyle\bigcup_{\alpha\in\mathcal{D}(\sigma)} \mathcal{L}(\varphi(\alpha)) \\
\mathcal{D}(\varphi(\sigma)) & \subseteq & \left(\displaystyle\bigcup_{\alpha\in\mathcal{D}(\sigma)} \mathcal{L}(\varphi(\alpha))\right) \cup \left(\displaystyle\bigcup_{\alpha\in\mathcal{L}(\sigma)} \mathcal{D}(\varphi(\alpha))\right)
\end{array}$$

Les mêmes résultats valent (trivialement) avec le type σ remplacé par une étiquette u.

Démonstration : immédiat par récurrence sur σ. □

On note $\mathcal{L}_g$ l'ensemble des variables libres génériques, $\mathcal{L}_n$ l'ensemble des variables libres non génériques, $\mathcal{D}_g$ l'ensemble des variables dangereuses génériques, et $\mathcal{D}_n$ l'ensemble des variables dangereuses non génériques :

$$\begin{array}{rclcrcl} \mathcal{L}_g(\sigma) & = & \mathcal{L}(\sigma) \cap \texttt{VarTypGen} & \qquad & \mathcal{D}_g(\sigma) & = & \mathcal{D}(\sigma) \cap \texttt{VarTypGen} \\ \mathcal{L}_n(\sigma) & = & \mathcal{L}(\sigma) \cap \texttt{VarTypNongen} & & \mathcal{D}_n(\sigma) & = & \mathcal{D}(\sigma) \cap \texttt{VarTypNongen} \end{array}$$

Les définitions de $\mathcal{L}$ et $\mathcal{D}$ ci-dessus ne tiennent pas compte des contraintes qui peuvent porter sur les étiquettes apparaissant dans le type τ. Par la suite, on a besoin d'une notion de variables libres (ou dangereuse) dans un type contraint σ / C, ou bien directement, ou bien par l'intermédiaire d'une contrainte. On parlera de variables récursivement libres ou récursivement dangereuses dans $\sigma \,/\, C$, par opposition aux variables directement libres ou directement dangereuses dans σ définies ci-dessus.

L'intuition est qu'une étiquette u doit être traitée comme un nœud dont les fils sont les types attachés à u dans l'ensemble de contraintes C. En particulier, pour que le typage des fermetures joue son rôle, il faut exprimer qu'une variable est récursivement libre dans $\sigma_1 \multimap\!\langle u \rangle\!\rightarrow \sigma_2 \,/\, C$ si elle est récursivement libre dans $\sigma \,/\, C$, pour un certain σ tel que la contrainte $\sigma \triangleleft u$ apparaisse dans C. Traduisant directement cette intuition, on définit :

$$\begin{array}{rcl} \mathcal{L}(\iota \,/\, C) & = & \emptyset \\ \mathcal{L}(t \,/\, C) & = & \{t\} \\ \mathcal{L}(\sigma_1 \multimap\!\langle u \rangle\!\rightarrow \sigma_2 \,/\, C) & = & \mathcal{L}(\sigma_1 \,/\, C) \cup \mathcal{L}(u \,/\, C) \cup \mathcal{L}(\sigma_2 \,/\, C) \\ \mathcal{L}(\sigma_1 \times \sigma_2 \,/\, C) & = & \mathcal{L}(\sigma_1 \,/\, C) \cup \mathcal{L}(\sigma_2 \,/\, C) \\ \mathcal{L}(\sigma\ \texttt{ref} \,/\, C) & = & \mathcal{L}(\sigma \,/\, C) \\ \mathcal{L}(\sigma\ \texttt{chan} \,/\, C) & = & \mathcal{L}(\sigma \,/\, C) \\ \mathcal{L}(\sigma\ \texttt{cont} \,/\, C) & = & \mathcal{L}(\sigma \,/\, C) \\ \mathcal{L}(u \,/\, C) & = & \{u\} \cup \bigcup\limits_{(\sigma \triangleleft u) \in C} \mathcal{L}(\sigma \,/\, C) \end{array}$$

De même pour les variables dangereuses :

$$\begin{array}{rcl} \mathcal{D}(\iota \,/\, C) & = & \emptyset \\ \mathcal{D}(t \,/\, C) & = & \emptyset \\ \mathcal{D}(\sigma_1 \multimap\!\langle u \rangle\!\rightarrow \sigma_2 \,/\, C) & = & \mathcal{D}(u \,/\, C) \\ \mathcal{D}(\sigma_1 \times \sigma_2 \,/\, C) & = & \mathcal{D}(\sigma_1 \,/\, C) \cup \mathcal{D}(\sigma_2 \,/\, C) \\ \mathcal{D}(\sigma\ \texttt{ref} \,/\, C) & = & \mathcal{L}(\sigma \,/\, C) \\ \mathcal{D}(\sigma\ \texttt{chan} \,/\, C) & = & \mathcal{L}(\sigma \,/\, C) \\ \mathcal{D}(\sigma\ \texttt{cont} \,/\, C) & = & \mathcal{L}(\sigma \,/\, C) \\ \mathcal{D}(u \,/\, C) & = & \bigcup\limits_{(\sigma \triangleleft u) \in C} \mathcal{D}(\sigma \,/\, C) \end{array}$$

Ces ensembles d'égalités ne forment pas une définition bien fondée. En effet, dans le cas $\mathcal{L}(u \,/\, C)$ ou $\mathcal{D}(u \,/\, C)$, un des types σ sur lesquels on fait l'union peut contenir comme sous-terme un type fonctionnel étiqueté par u. En d'autres termes, rien n'empêche les contraintes C d'être cycliques.

La "définition" ci-dessus doit en fait être lue comme un ensemble d'équations dont il faut trouver la plus petite solution. On voit facilement que les solutions sont exactement les points fixes de fonctionnelles croissantes bornées ; d'où l'existence d'une plus petite solution définissant $\mathcal{L}(\sigma \,/\, C)$ et $\mathcal{D}(\sigma \,/\, C)$.

On va maintenant donner une caractérisation plus facile à calculer de $\mathcal{L}$ et de $\mathcal{D}$, qui se révèle plus maniable pour faire des preuves. Pour tout entier n, on définit :

$$\begin{array}{rcl}
\mathcal{L}^0(\sigma) & = & \emptyset \\
\mathcal{L}^{n+1}(\iota \,/\, C) & = & \emptyset \\
\mathcal{L}^{n+1}(t \,/\, C) & = & \{t\} \\
\mathcal{L}^{n+1}(\sigma_1 \multimap\!\langle u\rangle\!\rightarrowtail \sigma_2 \,/\, C) & = & \mathcal{L}^n(\sigma_1 \,/\, C) \cup \mathcal{L}^n(u \,/\, C) \cup \mathcal{L}^n(\sigma_2 \,/\, C) \\
\mathcal{L}^{n+1}(\sigma_1 \times \sigma_2 \,/\, C) & = & \mathcal{L}^n(\sigma_1 \,/\, C) \cup \mathcal{L}^n(\sigma_2 \,/\, C) \\
\mathcal{L}^{n+1}(\sigma\ \texttt{ref} \,/\, C) & = & \mathcal{L}^n(\sigma \,/\, C) \\
\mathcal{L}^{n+1}(\sigma\ \texttt{chan} \,/\, C) & = & \mathcal{L}^n(\sigma \,/\, C) \\
\mathcal{L}^{n+1}(\sigma\ \texttt{cont} \,/\, C) & = & \mathcal{L}^n(\sigma \,/\, C) \\
\mathcal{L}^n(u \,/\, C) & = & \{u\} \cup \bigcup_{(\sigma \lhd u) \in C} \mathcal{L}^n(\sigma \,/\, C)
\end{array}$$

Pour les variables dangereuses, on prend :

$$\begin{array}{rcl}
\mathcal{D}^0(\sigma) & = & \emptyset \\
\mathcal{D}^{n+1}(\iota \,/\, C) & = & \emptyset \\
\mathcal{D}^{n+1}(t \,/\, C) & = & \emptyset \\
\mathcal{D}^{n+1}(\sigma_1 \multimap\!\langle u\rangle\!\rightarrowtail \sigma_2 \,/\, C) & = & \mathcal{D}^n(u \,/\, C) \\
\mathcal{D}^{n+1}(\sigma_1 \times \sigma_2 \,/\, C) & = & \mathcal{D}^n(\sigma_1 \,/\, C) \cup \mathcal{D}^n(\sigma_2 \,/\, C) \\
\mathcal{D}^{n+1}(\sigma\ \texttt{ref} \,/\, C) & = & \mathcal{L}(\sigma \,/\, C) \\
\mathcal{D}^{n+1}(\sigma\ \texttt{chan} \,/\, C) & = & \mathcal{L}(\sigma \,/\, C) \\
\mathcal{D}^{n+1}(\sigma\ \texttt{cont} \,/\, C) & = & \mathcal{L}(\sigma \,/\, C) \\
\mathcal{D}^n(u \,/\, C) & = & \bigcup_{(\sigma \lhd u) \in C} \mathcal{D}^n(\sigma \,/\, C)
\end{array}$$

Proposition 4.2 *Pour tout type contraint $\sigma \,/\, C$, on a*

$$\mathcal{L}(\sigma \,/\, C) = \bigcup_{n \geq 0} \mathcal{L}^n(\sigma \,/\, C) \qquad \mathcal{D}(\sigma \,/\, C) = \bigcup_{n \geq 0} \mathcal{D}^n(\sigma \,/\, C).$$

Démonstration : on montre que $\bigcup_{n\geq 0} \mathcal{L}^n(\sigma \,/\, C)$ et $\bigcup_{n\geq 0} \mathcal{D}^n(\sigma \,/\, C)$ vérifient les équations dont $\mathcal{L}$ et $\mathcal{D}$ sont les plus petits points fixes. Ceci établit l'inclusion $\subseteq$. Pour l'inclusion inverse, soient $\mathcal{L}'$ et $\mathcal{D}'$ des solutions quelconques de ces équations. On montre que $\mathcal{L}^n(\sigma \,/\, C) \subseteq \mathcal{L}'(\sigma \,/\, C)$ et que $\mathcal{D}^n(\sigma \,/\, C) \subseteq \mathcal{D}'(\sigma \,/\, C)$ pour tout n et pour tout σ, par récurrence sur n. Ceci établit les inclusions inverses, et donc les égalités annoncées. □

Voici l'inévitable lemme technique qui décrit l'effet d'une substitution sur l'ensemble des variables libres et l'ensemble des variables dangereuses dans un type.

Proposition 4.3 *Soient σ un type générique, C un ensemble de contraintes, et φ une substitution. On a :*

$$\begin{aligned}
\mathcal{L}(\varphi(\sigma) \,/\, C) &\supseteq \bigcup_{\alpha \in \mathcal{L}(\sigma/C)} \mathcal{L}(\varphi(\alpha)) \\
\mathcal{L}(\varphi(\sigma) \,/\, C) &= \bigcup_{\alpha \in \mathcal{L}(\sigma)} \mathcal{L}(\varphi(\alpha) \,/\, C) \\
\mathcal{D}(\varphi(\sigma) \,/\, C) &\supseteq \bigcup_{\alpha \in \mathcal{D}(\sigma)} \mathcal{L}(\varphi(\alpha) \,/\, C) \\
\mathcal{D}(\varphi(\sigma) \,/\, C) &\subseteq \left(\bigcup_{\alpha \in \mathcal{D}(\sigma)} \mathcal{L}(\varphi(\alpha) \,/\, C) \right) \cup \left(\bigcup_{\alpha \in \mathcal{L}(\sigma)} \mathcal{D}(\varphi(\alpha) \,/\, C) \right)
\end{aligned}$$

Démonstration : pour la première ligne, on montre l'inclusion correspondante pour $\mathcal{L}^n$, par récurrence sur n, et on conclut par la proposition 4.2. Pour les trois dernières lignes, on procède par récurrence structurelle sur σ, et application des égalités définissant $\mathcal{D}$ et $\mathcal{L}$. □

4.2.3 Équivalence de types contraints

La présentation des types en deux parties (une expression de type plus un ensemble de contraintes) présente un inconvénient : l'ensemble de contraintes étant partagé avec d'autres types (les types dans l'environnement de typage, en particulier), il peut contenir des contraintes qui ne portent pas sur des étiquettes accessibles à partir de l'expression de type. Ces contraintes supplémentaires ne changent pas les propriétés sémantiques du type ; mais elles peuvent changer des propriétés syntaxiques. Ainsi, si on substitue dans $\tau \,/\, C$ une variable α qui n'est pas libre dans $\tau \,/\, C$, on obtient un type contraint de la forme $\tau \,/\, C'$, avec C' éventuellement différent de C : la variable α peut en effet apparaître dans C, à l'intérieur de contraintes sur des étiquettes inaccessibles à partir de τ.

Exemple. Si on fait $t \mapsto \texttt{int}$ dans $\texttt{int} \multimap\!\langle u \rangle\!\rightarrowtail \texttt{int} \,/\, t \triangleleft v$, on obtient $\texttt{int} \multimap\!\langle u \rangle\!\rightarrowtail \texttt{int} \,/\, \texttt{int} \triangleleft v$, qui est syntaxiquement différent du type contraint initial. □

On va maintenant introduire une notion d'équivalence entre types contraints qui capture le fait que deux types contraints, bien que syntaxiquement différents, décrivent le même graphe de type. Soient σ un type et C un ensemble de contraintes. On définit la COMPOSANTE CONNEXE de σ dans C, notée $C \upharpoonright \sigma$, comme le sous-ensemble des contraintes de C qui portent sur une étiquette libre dans $\sigma \,/\, C$:

$$C \upharpoonright \sigma = \{(\sigma' \triangleleft u) \in C \mid u \in \mathcal{L}(\sigma \,/\, C)\}.$$

On définit de même $C \upharpoonright E$, la composante connexe de E, et $C \upharpoonright \{\sigma_1, \ldots, \sigma_n\}$, la composante connexe d'un ensemble de types.

Soient maintenant $\sigma_1 \,/\, C_1$ et $\sigma_2 \,/\, C_2$ deux types contraints. On dit que $\sigma_1 \,/\, C_1$ est ÉQUIVALENT à $\sigma_2 \,/\, C_2$, et on note $\sigma_1 \,/\, C_1 \equiv \sigma_2 \,/\, C_2$, si $\sigma_1 = \sigma_2$, et si $C_1 \upharpoonright \sigma_1 = C_2 \upharpoonright \sigma_2$. La relation $\equiv$ est clairement une relation d'équivalence.

Le reste de cette partie est consacré à quelques propriétés des composantes connexes et de la relation $\equiv$.

Proposition 4.4 *Pour tout type contraint $\sigma \,/\, C$, on a*

$$\mathcal{L}(\sigma \,/\, C) = \mathcal{L}(\sigma \,/\, (C \restriction \sigma)) \quad \textit{et} \quad \mathcal{D}(\sigma \,/\, C) = \mathcal{D}(\sigma \,/\, (C \restriction \sigma)).$$

En conséquence, si $\sigma_1 \,/\, C_1 \equiv \sigma_2 \,/\, C_2$, alors $\mathcal{L}(\sigma_1 \,/\, C_1) = \mathcal{L}(\sigma_2 \,/\, C_2)$ et $\mathcal{D}(\sigma_1 \,/\, C_1) = \mathcal{D}(\sigma_2 \,/\, C_2)$.

Démonstration : on montre les égalités analogues avec $\mathcal{L}^n$ et $\mathcal{D}^n$, pour tout n, par récurrence sur n. On conclut par la proposition 4.2. □

Proposition 4.5 *Soient $\sigma \,/\, C$ un type contraint, et φ une substitution. On a*

$$\varphi(C \restriction \sigma) \subseteq \varphi(C) \restriction \varphi(\sigma).$$

Démonstration : si une étiquette u est libre dans $\sigma \,/\, C$, alors l'étiquette $\varphi(u)$ est libre dans $\varphi(\sigma) \,/\, \varphi(C)$ (proposition 4.3). D'où le résultat. □

L'inclusion peut être stricte, car la substitution peut identifier deux étiquettes, et donc introduire de nouvelles contraintes dans la composante connexe d'un type.

Exemple. La composante connexe de $\texttt{int} \dashv\langle u\rangle\!\rightarrow \texttt{int}$ dans $\texttt{bool} \lhd u$, $\texttt{string} \lhd v$ est $\texttt{bool} \lhd u$. Mais si on applique la substitution $[v \mapsto u]$, la composante connexe devient $\texttt{bool} \lhd u$, $\texttt{string} \lhd u$. □

En conséquence, la relation d'équivalence de types contraints n'est pas stable par substitution.

Exemple. On a

$$\texttt{int} \dashv\langle u\rangle\!\rightarrow \texttt{int} \,/\, \texttt{bool} \lhd u, \texttt{string} \lhd v \quad \equiv \quad \texttt{int} \dashv\langle u\rangle\!\rightarrow \texttt{int} \,/\, \texttt{bool} \lhd u, \texttt{char} \lhd v.$$

Mais si on applique la substitution $[v \mapsto u]$ des deux côtés, on se retrouve avec deux types contraints qui ne sont pas équivalents. □

Il y a cependant une classe importante de substitutions qui commutent avec l'opération "composante connexe", et donc qui préservent l'équivalence entre types contraints : les substitutions qui sont des renommages.

Proposition 4.6 *Pour tout renommage θ, on a*

$$\theta(C \restriction \sigma) = \theta(C) \restriction \theta(\sigma).$$

En conséquence, $\sigma_1 \,/\, C_1 \equiv \sigma_2 \,/\, C_2$ implique $\theta(\sigma_1) \,/\, \theta(C_1) \equiv \theta(\sigma_2) \,/\, \theta(C_2)$.

Démonstration : par la proposition 4.5 appliquée à θ et à θ^{-1}, on a

$$C \restriction \sigma = \theta^{-1}(\theta(C \restriction \sigma)) \subseteq \theta^{-1}(\theta(C) \restriction \theta(\sigma)) \subseteq \theta^{-1}(\theta(C)) \restriction \theta^{-1}(\theta(\sigma)) = C \restriction \sigma.$$

D'où $\theta^{-1}(\theta(C) \restriction \theta(\sigma)) = C \restriction \sigma$, et l'égalité annoncée en appliquant θ des deux côtés. □

Dans les chapitres 1 et 3, on a utilisé abondamment la propriété suivante, vraie dans toutes les algèbres de types usuelles : si $\mathcal{L}(\tau)$ et $\mathrm{Dom}(\varphi)$ sont disjoints, alors $\varphi(\tau)$ est identique à τ. Cette propriété n'est pas vraie pour l'algèbre de types de ce chapitre : on peut avoir $\mathcal{L}(\sigma \,/\, C)$ et $\mathrm{Dom}(\varphi)$ disjoints, et pourtant $\varphi(\sigma) \,/\, \varphi(C)$ n'est pas équivalent à $\sigma \,/\, C$. En effet, la substitution φ peut "greffer" des contraintes supplémentaires sur des étiquettes libres dans $\sigma \,/\, C$.

Exemple. . Considérons le type $\sigma = \texttt{int} \multimap\!\langle u \rangle\!\rightarrow \texttt{int}$ sous les contraintes $C = \texttt{bool} \lhd v$. L'étiquette v n'est pas libre dans $\sigma \,/\, C$. Néanmoins, après substitution de v par u, on obtient le type contraint $\texttt{int} \multimap\!\langle u \rangle\!\rightarrow \texttt{int} \,/\, \texttt{bool} \lhd u$, qui n'est pas équivalent à $\sigma \,/\, C$. □

Néanmoins, la propriété attendue est vraie si la substitution est un renommage.

Proposition 4.7 *Soient θ un renommage et $\sigma \,/\, C$ un type contraint. Si* $\mathrm{Dom}(\theta) \cap \mathcal{L}(\sigma \,/\, C) = \emptyset$, *alors $\theta(\sigma) \,/\, \theta(C) \equiv \sigma \,/\, C$.*

Démonstration : on a $\theta(\sigma) = \sigma$, puisque, a fortiori, $\mathrm{Dom}(\theta) \cap \mathcal{L}(\sigma) = \emptyset$. D'autre part, par la proposition 4.6, $\theta(C) \restriction \theta(\sigma) = \theta(C \restriction \sigma)$. Soit $\sigma' \lhd u$ une contrainte de $C \restriction \sigma$. Toutes les variables libres dans $\sigma' \lhd u$ sont libres dans $\sigma \,/\, C$, et donc invariantes par θ. Donc $\theta(\sigma' \lhd u) = \sigma' \lhd u$. D'où $\theta(C) \restriction \theta(\sigma) = C \restriction \sigma$, et le résultat annoncé. □

4.2.4 Règles de typage

Dans le système indirect, le prédicat de typage est de la forme $E \vdash a : \tau \,/\, C$, qu'il faut lire : "dans l'environnement E, l'expression a a le type non générique τ, compte tenu des contraintes sur les étiquettes apparaissant dans C". L'environnement E est une application finie des identificateurs dans les types génériques. Les variables génériques dans $E(x)$ sont traitées comme universellement quantifiées.

Les règles de typage sont essentiellement une reformulation des règles de la partie 3.2.3, avec les types directs remplacés par des types indirects plus un ensemble de contraintes, et les schémas de types remplacés par des types génériques. Les différences essentielles sont, d'une part, l'étape de généralisation de la règle `let`, et d'autre part l'introduction d'une règle permettant de simplifier les ensembles de contraintes.

$$\frac{\tau \leq E(x) \,/\, C}{E \vdash x : \tau \,/\, C}$$

La relation d'instanciation d'un type générique en un type non générique est définie comme suit : $\tau \leq \sigma \,/\, C$ si et seulement si il existe une substitution $\varphi : \mathcal{L}_g(E(x)) \Rightarrow \texttt{TypNongen} \cup \texttt{EtiqNongen}$ telle que $\varphi(\sigma) = \tau$ et $\varphi(C) \subseteq C$. L'instanciation d'un type générique se traduit donc par une substitution des variables génériques de ce type par des types et des étiquettes non génériques. Cette substitution s'applique également aux contraintes : C peut contenir des contraintes portant sur des étiquettes génériques appartenant à σ. La condition $\varphi(C) \subseteq C$ traduit le fait que C garde correctement trace de ces contraintes après l'instanciation.

$$\frac{E + f \mapsto (\tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2 \,/\, C \qquad (E(y) \lhd u_n) \in C \text{ pour tout } y \in \mathcal{I}(f \texttt{ where } f(x) = a)}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2 \,/\, C}$$

La règle de typage des fonctions exige donc qu'à l'étiquette u_n soient attachés, dans C, les types des identificateurs libres dans la fonction.

$$\frac{E \vdash a_1 : \tau_2 \multimap\!\langle u_n \rangle\!\rightarrow \tau_1 \,/\, C \qquad E \vdash a_2 : \tau_2 \,/\, C}{E \vdash a_1(a_2) : \tau_1 \,/\, C} \qquad \frac{E \vdash a_1 : \tau_1 \,/\, C \qquad E \vdash a_2 : \tau_2 \,/\, C}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2 \,/\, C}$$

$$\frac{E \vdash a_1 : \tau_1 \,/\, C_1 \qquad (\sigma, C) = \texttt{Gen}(\tau_1, C_1, E) \qquad E + x \mapsto \sigma \vdash a_2 : \tau_2 \,/\, C}{E \vdash \texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2 : \tau_2 \,/\, C}$$

La relation de généralisation est définie comme suit : $(\sigma, C) = \texttt{Gen}(\tau_1, C_1, E)$ si et seulement si il existe un renommage θ de $\mathcal{L}(\tau_1 \,/\, C_1) \setminus \mathcal{D}(\tau_1 \,/\, C_1) \setminus \mathcal{L}_n(E) \setminus \mathcal{D}(E \,/\, C_1)$ en variables génériques non libres dans E ni dans C_1 tel que $\sigma = \theta(\tau_1)$ et $C = \theta(C_1)$.

En d'autres termes, le domaine de θ est constitué des variables non génériques dans le type τ_1 qu'on veut généraliser. Pour ce faire, on les renomme en des variables génériques. Le renommage est également appliqué à l'ensemble de contraintes courant.

À la différence du système du chapitre 3, on se permet ici de généraliser des variables qui sont indirectement libres dans $E \,/\, C_1$, pourvu qu'elles ne soient pas directement libres dans E, ni dangereuses dans $E \,/\, C_1$. C'est-à-dire, on ignore les variables non dangereuses qui sont capturées par les types de fermetures de E. (Voir l'exemple ci-dessous.)

Les règles pour les constantes et les opérateurs sont sans surprises :

$$\frac{\tau \leq \texttt{TypCst}(cst) \,/\, C}{E \vdash cst : \tau \,/\, C}$$

$$\frac{\tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2 \leq \texttt{TypOp}(op) \,/\, C \qquad E \vdash a : \tau_1 \,/\, C}{E \vdash op(a) : \tau_2 \,/\, C}$$

La dernière règle est une règle de simplification, qui permet d'effacer des contraintes qui ne servent à rien dans la dérivation de typage, et de les remplacer par d'autres contraintes :

$$\frac{E' \vdash a : \tau' \,/\, C' \qquad \tau \,/\, C \equiv \tau' \,/\, C' \qquad E\,|_{\,\mathcal{I}(a)} \,/\, C \equiv E'\,|_{\,\mathcal{I}(a)} \,/\, C'}{E \vdash a : \tau \,/\, C}$$

L'intuition de cette règle est que seule les contraintes qui sont dans la composante connexe de τ et de la restriction de E aux identificateurs libres dans a ont une influence sur le typage de l'expression. Les autres contraintes peuvent librement être effacées, ou remplacées par d'autres contraintes du même genre.

Les types des opérateurs primitifs sur les références, les canaux, et les continuations sont ceux de la partie 3.2.3, ré-exprimés avec des étiquettes au lieu de variables d'expansion, et des variables génériques au lieu de variables quantifiées.

$$\begin{aligned}
\texttt{TypOp}(\texttt{ref}) &= t_g \multimap\!\langle u_g \rangle\!\rightarrow t_g\ \ \texttt{ref} \\
\texttt{TypOp}(!) &= t_g\ \ \texttt{ref} \multimap\!\langle u_g \rangle\!\rightarrow t_g \\
\texttt{TypOp}(:=) &= t_g\ \ \texttt{ref} \times t_g \multimap\!\langle u_g \rangle\!\rightarrow \texttt{unit} \\
\\
\texttt{TypOp}(\texttt{newchan}) &= \texttt{unit} \multimap\!\langle u_g \rangle\!\rightarrow t_g\ \ \texttt{chan} \\
\texttt{TypOp}(?) &= t_g\ \ \texttt{chan} \multimap\!\langle u_g \rangle\!\rightarrow t_g \\
\texttt{TypOp}(!) &= t_g\ \ \texttt{chan} \times t_g \multimap\!\langle u_g \rangle\!\rightarrow \texttt{unit}
\end{aligned}$$

$$\begin{aligned} \texttt{TypOp(callcc)} &= (t_g\ \texttt{cont} \dashv\langle u_g\rangle\rightarrowtail t_g) \dashv\langle u'_g\rangle\rightarrowtail t_g \\ \texttt{TypOp(throw)} &= t_g\ \texttt{cont} \times t_g \dashv\langle u_g\rangle\rightarrowtail t'_g \end{aligned}$$

Exemple. On reprend l'exemple de non-conservativité de la partie 4.1.2.

$$\lambda\texttt{f. let id} = \lambda\texttt{y. either(f)}(\lambda\texttt{z. y; z});\ \texttt{y in id(id)}$$

Le typage principal de la partie gauche du `let` débouche sur :

$$[\texttt{f} : t_n \dashv\langle u_n\rangle\rightarrowtail t_n] \vdash (\lambda\texttt{y. either(f)}(\lambda\texttt{z. y; z})\texttt{; y}) : t'_n \dashv\langle u'_n\rangle\rightarrowtail t'_n \,/\, t'_n \lhd u_n,\ (t_n \dashv\langle u_n\rangle\rightarrowtail t_n) \lhd u'_n.$$

Les variables généralisables sont t'_n et u'_n. En effet, t'_n n'est pas directement libre dans l'environnement de typage, bien qu'elle soit libre dans une contrainte accessible depuis cet environnement. On peut donc généraliser t'_n en t_g et u'_n en u_g, et l'auto-application `id(id)` est bien typée. □

4.2.5 Propriétés du typage

Dans cette partie, on montre que le prédicat de typage est, sous certaines conditions, stable par substitution de variables de types et par ajout de contraintes.

Proposition 4.8 (Stabilité du typage par substitution) *Soient a une expression, τ un type non générique, E un environnement de typage, C un ensemble de contraintes. Pour toute substitution φ : $\texttt{VarNongen} \Rightarrow \texttt{TypNongen} \cup \texttt{EtiqNongen}$ de variables non génériques par des types non génériques, si on a $E \vdash a : \tau \,/\, C$, alors on a $\varphi(E) \vdash a : \varphi(\tau) \,/\, \varphi(C)$.*

Proposition 4.9 (Stabilité du typage par ajout de contraintes) *Soient a une expression, τ un type, E un environnement de typage, C un ensemble de contraintes tels que $E \vdash a : \tau \,/\, C$. Soit C' un ensemble de contraintes où n'apparaissent aucune des variables génériques libres dans $E(y)$ quel que soit y libre dans a. C'est-à-dire :*

$$(\{u\} \cup \mathcal{L}(\sigma)) \cap \mathcal{L}_g(E(y)) = \emptyset \text{ pour tous } (\sigma \lhd u) \in C',\ y \in \mathcal{I}(a).$$

Alors $E \vdash a : \tau \,/\, C \cup C'$.

Les deux propositions se prouvent simultanément, par récurrence sur la hauteur de la dérivation de $E \vdash a : \tau \,/\, C$, et par cas sur la dernière règle employée. La preuve étant fort longue, je préfère la présenter de la manière suivante : je prouve chaque proposition séparément, en supposant l'autre vraie. Je montre de plus que les dérivations obtenues (de $\varphi(E) \vdash a : \varphi(\tau) \,/\, \varphi(C)$ pour la proposition 4.8, de $E \vdash a : \tau \,/\, C \cup C'$ pour la proposition 4.9) sont de même hauteur que la dérivation initiale (de $E \vdash a : \tau \,/\, C$). Ceci assure que la récurrence simultanée est bien fondée, parce que l'autre proposition est toujours appliquée à des dérivations strictement moins hautes que la dérivation d'origine.

Démonstration : de la proposition 4.8. On vérifie facilement pour chaque cas que la dérivation de $\varphi(E) \vdash a : \varphi(\tau) \,/\, \varphi(C)$ construite a bien la même hauteur que la dérivation initiale de $E \vdash a : \tau \,/\, C$.

• **Règle d'instanciation.** On a $a = x$ et $\tau \leq E(x) \,/\, C$. Soit $\psi : \mathcal{L}_g(E(x)) \Rightarrow \texttt{TypNongen} \cup \texttt{EtiqNongen}$ une substitution telle que $\psi(E(x)) = \tau$ et $\psi(C) \subseteq C$. On définit une substitution θ par $\theta(\alpha_g) = \varphi(\psi(\alpha_g))$ pour toute variable générique α_g. On a :

$$\begin{array}{rcccll} \theta(\varphi(\alpha_g)) & = & \theta(\alpha_g) & = & \varphi(\psi(\alpha_g)) & \text{pour toute } \alpha_g \text{ générique} \\ \theta(\varphi(\alpha_n)) & = & \varphi(\alpha_n) & = & \varphi(\psi(\alpha_n)) & \text{pour toute } \alpha_n \text{ non générique} \end{array}$$

D'où $\theta \circ \varphi = \varphi \circ \psi$. Donc, $\theta(\varphi(E(x))) = \varphi(\psi(E(x))) = \varphi(\tau)$, et de même $\theta(\varphi(C)) = \varphi(\psi(C)) \subseteq \varphi(C)$. Enfin, comme φ n'opère que sur des variables non génériques et n'introduit pas de nouvelles variables génériques, on a $\mathcal{L}_g(\varphi(E(x))) = \mathcal{L}_g(E(x))$, et donc $\theta : \mathcal{L}_g(\varphi(E(x))) \Rightarrow \texttt{TypNongen} \cup \texttt{EtiqNongen}$. Ceci établit que $\varphi(\tau) \leq \varphi(E(x)) \,/\, \varphi(C)$. On peut donc dériver $\varphi(E) \vdash x : \varphi(\tau) \,/\, \varphi(C)$.

• **Règle du `where`.** La dérivation est de la forme :

$$\frac{\begin{array}{c} E + f \mapsto (\tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2 \,/\, C \\ (E(y) \triangleleft u_n) \in C \text{ pour tout } y \in \mathcal{I}(f \texttt{ where } f(x) = a) \end{array}}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2 \,/\, C}$$

Par hypothèse de récurrence appliquée à a, on obtient une preuve de :

$$\varphi(E + f \mapsto (\tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2) + x \mapsto \tau_1) \vdash a : \varphi(\tau_2) \,/\, \varphi(C).$$

Par définition de la substitution sur les contraintes, on a $(\varphi(\sigma) \triangleleft \varphi(u_n)) \in \varphi(C)$ dès que $(\sigma \triangleleft u_n) \in C$. C'est en particulier vrai si $\sigma = E(y)$ avec y libre dans a. On peut donc dériver le résultat attendu :

$$\varphi(E) \vdash (f \texttt{ where } f(x) = a) : \varphi(\tau_1) \multimap\!\langle \varphi(u_n) \rangle\!\rightarrow \varphi(\tau_2) \,/\, \varphi(C).$$

• **Règle du `let`.** La dérivation est de la forme :

$$\frac{E \vdash a_1 : \tau_1 \,/\, C_1 \qquad (\sigma, C) = \texttt{Gen}(\tau_1, C_1, E) \qquad E + x \mapsto \sigma \vdash a_2 : \tau_2 \,/\, C}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2 \,/\, C}$$

Notons $\{\alpha_1 \ldots \alpha_k\} = \mathcal{L}(\tau \,/\, C_1) \setminus \mathcal{D}(\tau \,/\, C_1) \setminus \mathcal{L}_n(E) \setminus \mathcal{D}(E \,/\, C_1)$. (Les α_i sont non génériques. On omet l'indice n pour plus de lisibilité.) Soit $\theta : \{\alpha_1 \ldots \alpha_k\} \Leftrightarrow \texttt{VarTypGen}$ le renommage tel que $\sigma = \theta(\tau_1)$ et $C = \theta(C_1)$. Soient $\beta_1 \ldots \beta_k$ des variables non génériques, deux à deux distinctes, non libres dans E, non libres dans C_1, et hors de portée de φ, avec β_i de la même sorte que α_i pour tout i. On définit la substitution

$$\psi = \varphi \circ [\alpha_1 \mapsto \beta_1 \ldots \alpha_k \mapsto \beta_k].$$

On a $\psi(E) = \varphi(E)$, puisque les α_i ne sont pas libres dans E. On applique deux fois l'hypothèse de récurrence : à la prémisse de gauche, avec la substitution ψ ; et à la prémisse de droite, avec la substitution φ. Il vient des preuves de :

$$\psi(E) \vdash a_1 : \psi(\tau_1) \,/\, \psi(C_1) \qquad\qquad \varphi(E) + x \mapsto \varphi(\sigma) \vdash a_2 : \varphi(\tau_2) \,/\, \varphi(C).$$

Notons $V = \mathcal{L}(\psi(\tau_1) \,/\, \psi(C_1)) \setminus \mathcal{D}(\psi(\tau_1) \,/\, \psi(C_1)) \setminus \mathcal{L}_n(\psi(E)) \setminus \mathcal{D}(\psi(E) \,/\, \psi(C_1))$. On va maintenant montrer que $V = \{\beta_1 \dots \beta_k\}$. Par construction de ψ et des β_i, on a $\psi(\alpha_i) = \beta_i$, et $\beta_i \in \mathcal{L}(\psi(\alpha))$ pour une certaine variable α implique $\alpha = \alpha_i$.

On fixe i. Comme α_i est libre dans $\tau_1 \,/\, C_1$, on a β_i libre dans $\psi(\tau_1) \,/\, \psi(C_1)$ (prop. 4.3, cas 1). Comme α_i n'est pas libre dans E, on a $\beta_i \notin \mathcal{L}_n(\psi(E))$. Dans le cas contraire, on aurait $\beta_i \in \mathcal{L}(\psi(\alpha))$ pour un certain $\alpha \in \mathcal{L}(E)$ (prop 4.1, cas 1) ; mais par définition de ψ, cet α est forcément α_i, qui n'est pas libre dans E. De plus, $\beta_i \notin \mathcal{D}(\psi(\tau_1) \,/\, \psi(C_1))$. Dans le cas contraire, on aurait ou bien $\beta_i \in \mathcal{L}(\psi(\alpha) \,/\, \psi(C_1))$ pour un certain α dans $\mathcal{D}(\tau_1)$, ou bien $\beta_i \in \mathcal{D}(\psi(\alpha) \,/\, \psi(C_1))$ pour un certain α dans $\mathcal{L}(\tau_1)$ (prop 4.3, cas 4). La première possibilité est exclue car, par construction des β_i, seule $\alpha = \alpha_i$ conviendrait ; mais α_i n'est pas dangereuse dans $\tau_1 \,/\, C_1$, et donc a fortiori elle n'est pas directement dangereuse dans τ_1. La seconde possibilité conduit de même à $\alpha = \alpha_i$, mais

$$\mathcal{D}(\psi(\alpha_i) \,/\, \psi(C_1)) = \mathcal{D}(\beta_i \,/\, \psi(C_1)) = \emptyset$$

d'où contradiction. On montre de même que $\beta_i \notin \mathcal{D}(\psi(E) \,/\, \psi(C_1))$. On conclut donc que

$$\{\beta_1 \dots \beta_k\} \subseteq V.$$

On montre maintenant l'inclusion inverse. Soit β une variable non générique, libre dans $\psi(\tau_1) \,/\, \psi(C_1)$, et qui n'est pas une des β_i. Soit $\alpha \in \mathcal{L}(\tau_1 \,/\, C_1)$ une variable non générique telle que $\beta \in \mathcal{L}(\psi(\alpha))$. La variable α ne peut être une des α_i, car sinon β serait une des β_i. Donc ou bien α est directement libre dans E, ou bien α est dangereuse dans $\tau_1 \,/\, C_1$, ou bien α est dangereuse dans $E \,/\, C_1$. Si α est libre dans E, alors β est libre dans $\psi(E)$ (prop 4.1, cas 1). Si α est dangereuse dans $\tau_1 \,/\, C$, alors β est dangereuse dans $\psi(\tau_1) \,/\, \psi(C_1)$ (prop 4.3, cas 3). Enfin, si α est dangereuse dans $E \,/\, C$, alors β est dangereuse dans $\psi(E) \,/\, \psi(C_1)$ (prop 4.3, cas 3). Dans les trois cas, $\beta \notin V$. D'où l'inclusion inverse recherchée.

On définit le renommage $\xi = [\beta_1 \mapsto \theta(\alpha_1), \dots, \beta_k \mapsto \theta(\alpha_k)]$. On a bien $\xi : \{\beta_1, \dots, \beta_k\} \Leftrightarrow$ `VarTypGen`. On a, pour tout i :

$$\begin{array}{rcll} \xi(\psi(\alpha_i)) & = & \xi(\beta_i) & \text{par définition de } \psi \\ & = & \theta(\alpha_i) & \text{par définition de } \xi \\ & = & \varphi(\theta(\alpha_i)) & \text{car } \varphi \text{ n'affecte pas les variables génériques.} \end{array}$$

D'autre part, pour toute variable α qui n'est ni une α_i, ni une β_i :

$$\begin{array}{rcll} \xi(\psi(\alpha)) & = & \xi(\varphi(\alpha)) & \text{par définition de } \psi \\ & = & \varphi(\alpha) & \text{car } \alpha \notin \{\beta_1 \dots \beta_k\} \text{ et les } \beta_i \text{ sont hors de portée de } \varphi \\ & = & \varphi(\theta(\alpha)) & \text{car } \theta(\alpha) = \alpha \text{ puisque } \alpha \notin \{\alpha_1 \dots \alpha_k\}. \end{array}$$

Comme les β_i ne sont libres ni dans τ_1 ni dans C_1, il s'ensuit que $\xi(\psi(\tau_1)) = \varphi(\theta(\tau_1)) = \varphi(\tau)$ et $\xi(\psi(C_1)) = \varphi(\theta(C_1)) = \varphi(C)$. On a donc établi que :

$$(\varphi(\sigma), \varphi(C)) = \texttt{Gen}(\psi(\tau_1), \psi(C_1), \varphi(E))$$

Combinant ce fait avec les deux dérivations obtenues par application de l'hypothèse de récurrence :

$$\varphi(E) \vdash a_1 : \psi(\tau_1) \,/\, \psi(C_1) \qquad \varphi(E) + x \mapsto \varphi(\sigma) \vdash a_2 : \varphi(\tau_2) \,/\, \varphi(C),$$

on déduit, par application de la règle `let`, le résultat attendu :

$$\varphi(E) \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \varphi(\tau_2)\ /\ \varphi(C).$$

• **Règle de simplification.** La dérivation initiale est de la forme :

$$\frac{E \vdash a : \tau\ /\ C \qquad \tau\ /\ C \equiv \tau\ /\ C' \qquad E\,|_{\mathcal{I}(a)}\ /\ C \equiv E'\,|_{\mathcal{I}(a)}\ /\ C'}{E' \vdash a : \tau\ /\ C'}$$

La preuve de ce cas est un peu délicate, parce que on n'a pas forcément $\varphi(\tau)\ /\ \varphi(C) \equiv \varphi(\tau)\ /\ \varphi(C')$ et $\varphi(E\,|_{\mathcal{I}(a)})\ /\ \varphi(C) \equiv \varphi(E'\,|_{\mathcal{I}(a)})\ /\ \varphi(C')$. Par exemple, on a

$$\texttt{int} \multimap\!\langle u\rangle\!\rightarrowtail \texttt{int}\ /\ \texttt{bool} \lhd v \quad\equiv\quad \texttt{int} \multimap\!\langle u\rangle\!\rightarrowtail \texttt{int}\ /\ \emptyset,$$

mais cette équivalence n'est plus vraie si on substitue v par u. On va s'en tirer par renommage massif. Soit U l'ensemble des étiquettes libres dans $\tau\ /\ C$ ou dans $E\,|_{\mathcal{I}(a)}\ /\ C$. On décompose C en $C_0 \cup C_1$ et C' en $C_0 \cup C_1'$, où C_0 ne contraint que des étiquettes $u_1 \dots u_k$ appartenant à U, alors que les étiquettes $v_1 \dots v_m$ contraintes par C_1 et les étiquettes $v_1' \dots v_p'$ contraintes par C_1' n'appartiennent pas à U. Soient $w_1 \dots w_m$ et $w_1' \dots w_p'$ de nouvelles étiquettes, toutes différentes, hors de portée de φ, et n'appartenant pas à U. On définit deux substitutions θ et ψ comme suit :

$$\begin{array}{rcl} \theta & = & [v_1' \mapsto w_1', \dots, v_p' \mapsto w_p'] \\ \psi & = & \varphi + v_1 \mapsto w_1 + \dots + v_m \mapsto w_m + w_1' \mapsto \varphi(v_1') + \dots + w_p' \mapsto \varphi(v_p') \end{array}$$

Par construction, aucune des variables libres dans $E\,|_{\mathcal{I}(a)}$ n'apparaît dans les contraintes de $\theta(C_1')$. Appliquant la proposition 4.9, il vient une dérivation de

$$E \vdash a : \tau\ /\ C \cup \theta(C_1')$$

de même hauteur que la dérivation de $E \vdash a : \tau\ /\ C$. On peut donc appliquer l'hypothèse de récurrence à la dérivation obtenue. Il vient une dérivation de

$$\psi(E) \vdash a : \psi(\tau)\ /\ \psi(C \cup \theta(C_1'))$$

elle aussi de même hauteur que la dérivation de $E \vdash a : \tau\ /\ C$. Comme les v_i et les w_j' ne sont pas libres dans τ, on a $\psi(\tau) = \varphi(\tau)$. De même, $\psi(E\,|_{\mathcal{I}(a)}) = \varphi(E'\,|_{\mathcal{I}(a)})$. Enfin, $\psi(C_0) = \varphi(C_0)$ puisque toutes les étiquettes libres dans une des contraintes de C_0 sont dans U. D'autre part, $\psi(\theta(C_1')) = \varphi(C_1')$ par construction. Donc :

$$\psi(C \cup \theta(C_1')) = \psi(C_0 \cup C_1 \cup \theta(C_1')) = \varphi(C_0) \cup \psi(C_1) \cup \varphi(C_1') = \varphi(C') \cup \psi(C_1).$$

On remarque de plus que les contraintes de $\psi(C_1)$ portent uniquement sur les étiquettes $w_1 \dots w_m$, qui ne sont pas libres dans $\varphi(\tau)\ /\ \varphi(C') \cup \psi(C_1)$, ni dans $\varphi(E'\,|_{\mathcal{I}(a)})\ /\ \varphi(C') \cup \psi(C_1)$, puisque les antécédents de ces étiquettes, $v_1 \dots v_m$, ne sont pas libres dans $E'\,|_{\mathcal{I}(a)}\ /\ C'$, ni dans $\tau\ /\ C'$. On a donc établi que :

$$\begin{array}{rcl} \psi(E) \vdash a : \psi(\tau)\ /\ \varphi(C') \cup \psi(C_1) & & \\ \psi(\tau)\ /\ \varphi(C') \cup \psi(C_1) & \equiv & \varphi(\tau)\ /\ \varphi(C') \\ \psi(E\,|_{\mathcal{I}(a)})\ /\ \varphi(C') \cup \psi(C_1) & \equiv & \varphi(E'\,|_{\mathcal{I}(a)})\ /\ \varphi(C') \end{array}$$

De ces prémisses, on peut déduire, par la règle de simplification :

$$\varphi(E') \vdash a : \varphi(\tau) \,/\, \varphi(C')$$

et la dérivation ainsi construite est de même hauteur que la dérivation initiale de $E' \vdash a : \tau \,/\, C$. □

Démonstration : (de la proposition 4.9). On donne les cas qui ne s'ensuivent pas immédiatement de l'hypothèse de récurrence. Là encore, on vérifie sans peine pour chaque cas que la dérivation de $E \vdash a : \tau \,/\, C'$ obtenue a bien la même hauteur que la dérivation initiale de $E' \vdash a : \tau \,/\, C'$.

• **Règle d'instanciation.** La dérivation est de la forme

$$\frac{\tau \leq E(x) \,/\, C}{E \vdash x : \tau \,/\, C}$$

Soit $\varphi : \mathcal{L}_g(E(x)) \Rightarrow \texttt{TypNongen} \cup \texttt{EtiqNongen}$ la substitution telle que $\varphi(\sigma) = \tau$ et $\varphi(C) \subseteq C$. Comme φ n'affecte que des variables génériques libres dans E, on a $\varphi(C') = C'$ par hypothèse sur C', et donc

$$\varphi(C \cup C') = \varphi(C) \cup \varphi(C') = \varphi(C) \cup C' \subseteq C \cup C'.$$

On a donc bien $\tau \leq E(x) \,/\, C \cup C'$, et la règle d'instanciation permet de conclure $E \vdash x : \tau \,/\, C \cup C'$.

• **Règle du `let`.** La dérivation initiale est de la forme :

$$\frac{E \vdash a_1 : \tau_1 \,/\, C_1 \qquad (\sigma, C) = \texttt{Gen}(\tau_1, C_1, E) \qquad E + x \mapsto \sigma \vdash a_2 : \tau_2 \,/\, C}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2 \,/\, C}$$

Notons $\{\alpha_1 \ldots \alpha_k\} = \mathcal{L}(\tau_1 \,/\, C_1) \setminus \mathcal{D}(\tau_1 \,/\, C_1) \setminus \mathcal{L}_n(E)$. On peut se ramener au cas où les α_i n'apparaissent pas dans C', en renommant les α_i en de nouvelles variables β_i dans la première prémisse. La dérivation reste valide, comme on l'a montré dans le cas `let` de la proposition 4.8.

On applique l'hypothèse de récurrence aux deux prémisses. Il vient des preuves de :

$$E \vdash a_1 : \tau_1 \,/\, C_1 \cup C' \qquad\qquad E + x \mapsto \sigma \vdash a_2 : \tau_2 \,/\, C \cup C'$$

Pour conclure, il suffit de montrer que

$$\texttt{Gen}(\tau_1, C_1 \cup C', E) = (\sigma, C \cup C').$$

Tout d'abord, on a

$$\mathcal{L}(\tau_1 \,/\, C_1) \setminus \mathcal{D}(\tau_1 \,/\, C_1 \cup C') \setminus \mathcal{L}_n(E) \setminus \mathcal{D}(E \,/\, C_1 \cup C') = \{\alpha_1 \ldots \alpha_k\}.$$

En effet, on voit immédiatement que $\mathcal{D}(\tau_1 \,/\, C_1 \cup C')$ contient au moins $\mathcal{D}(\tau_1 \,/\, C_1)$. Et d'autre part α_i n'appartient pas à $\mathcal{D}(\tau_1 \,/\, C_1 \cup C')$, puisque α_i n'appartient pas à $\mathcal{D}(\tau_1 \,/\, C_1)$, et puisque α_i n'apparaît pas dans C'. Même raisonnement pour $\mathcal{D}(E \,/\, C_1 \cup C')$.

Soit alors $\theta : \{\alpha_1 \dots \alpha_k\} \Leftrightarrow \texttt{VarTypGen}$ le renommage tel que $\sigma = \theta(\tau_1)$ et $C = \theta(C_1)$. Comme les α_i n'apparaissent pas dans C', on a $\theta(C') = C'$, et donc $\theta(C_1 \cup C') = \theta(C_1) \cup C' = C \cup C'$. On a donc bien, comme annoncé :

$$\texttt{Gen}(\tau_1, C_1 \cup C', E) = (\sigma, C \cup C').$$

Combinant ce fait avec les deux dérivations obtenues par application de l'hypothèse de récurrence, on conclut, par la règle `let` :

$$E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2 \,/\, C \cup C'.$$

C'est le résultat désiré.

• **Règle de simplification.** La dérivation initiale se conclut par :

$$\frac{E_1 \vdash a : \tau \,/\, C_1 \qquad \tau \,/\, C_1 \equiv \tau \,/\, C \qquad E_1 \mid_{\mathcal{I}(a)} / C_1 \equiv E \mid_{\mathcal{I}(a)} / C}{E \vdash a : \tau \,/\, C}$$

On applique l'hypothèse de récurrence à la prémisse $E_1 \vdash a : \tau \,/\, C_1$. Il vient une preuve de $E_1 \vdash a : \tau \,/\, C_1 \cup C'$. On vérifie facilement que $\tau \,/\, C_1 \equiv \tau \,/\, C$ entraîne $\tau \,/\, C_1 \cup C' \equiv \tau \,/\, C \cup C'$, et de même pour $E \mid_{\mathcal{I}(a)}$ et $E_1 \mid_{\mathcal{I}(a)}$. Appliquant la règle de simplification à ces prémisses, il vient $E \vdash a : \tau \,/\, C \cup C'$. □

Proposition 4.10 (Stabilité du typage par substitution, bis) *Soient a une expression, τ un type non générique, E un environnement de typage, C un ensemble de contraintes. Pour toute substitution $\varphi : \texttt{VarNongen} \Rightarrow \texttt{TypNongen} \cup \texttt{EtiqNongen}$ de variables non génériques par des types non génériques, si on a $E \vdash a : \tau \,/\, C$ et $\varphi(C) \subseteq C$, alors on a $\varphi(E) \vdash a : \varphi(\tau) \,/\, C$.*

Remarque. La proposition 4.10 n'est hélas pas une conséquence des propositions 4.8 et 4.9 : même si $\varphi(C) \subseteq C$, l'ensemble $C' = C \backslash \varphi(C)$ ne remplit pas forcément les hypothèses de la proposition 4.9. □

Démonstration : la preuve est très similaire à celle de la proposition 4.8. La seule différence est pour la règle d'instanciation.

• **Règle d'instanciation.** On a $a = x$ et $\tau \leq E(x) \,/\, C$. Soit $\psi : \mathcal{L}_g(E(x)) \Rightarrow \texttt{TypNongen} \cup \texttt{EtiqNongen}$ une substitution telle que $\psi(E(x)) = \tau$ et $\psi(C) \subseteq C$. On définit une substitution θ par $\theta(\alpha_g) = \varphi(\psi(\alpha_g))$ pour toute α_g générique. On a :

$$\begin{array}{rcccl} \theta(\varphi(\alpha_g)) & = & \theta(\alpha_g) & = & \varphi(\psi(\alpha_g)) \\ \theta(\varphi(\alpha_n)) & = & \varphi(\alpha_n) & = & \varphi(\psi(\alpha_n)) \end{array}$$

D'où $\theta \circ \varphi = \varphi \circ \psi$. Donc, $\theta(\varphi(E(x))) = \varphi(\psi(E(x))) = \varphi(\tau)$. De plus, $\theta(C) = \varphi(\psi(C)) \subseteq \varphi(C) \subseteq C$. Enfin, comme φ n'opère que sur des variables non génériques et n'introduit pas de nouvelles variables génériques, on a $\mathcal{L}_g(\varphi(E(x))) = \mathcal{L}_g(E(x))$, et donc $\theta : \mathcal{L}_g(\varphi(E(x))) \Rightarrow \texttt{TypNongen} \cup \texttt{EtiqNongen}$. Ceci établit que $\varphi(\tau) \leq \varphi(E(x)) \,/\, C$. On peut donc dériver $\varphi(E) \vdash x : \varphi(\tau) \,/\, C$. □

4.3 Preuves de correction du typage indirect

Dans cette partie, on montre que le système de types indirect proposé dans ce chapitre est correct pour chacune des trois sémantiques données au chapitre 2 : celle avec les références, celle avec les canaux, et celle avec les continuations.

4.3.1 Références

Les prédicats de typage sémantique qu'on va utiliser sont essentiellement les mêmes que ceux de la partie 3.3.1, exprimés en termes de types indirects et de types génériques au lieu de types directs et de schémas de types. Les TYPES D'ÉTATS MÉMOIRE, notés S, associent maintenant un type contraint (une expression de type plus un ensemble de contraintes) à chaque adresse mémoire utilisée.

$$\text{Type d'état mémoire :} \quad S \quad ::= \quad [\ell_1 \mapsto \tau_1 \,/\, C_1, \ldots, \ell_n \mapsto \tau_n \,/\, C_n]$$

Comme dans la partie 3.3.1, on emploie des types d'états mémoire pour assurer que toutes les références à une adresse mémoire ℓ ont le même type monomorphe τ `ref` $/\, C$, où $\tau \,/\, C$ est identique à $S(\ell)$. Ceci écarte toute possibilité d'utilisation incohérente de l'adresse ℓ. En fait, on relâche légèrement la condition ci-dessus, et on se contente d'exiger que $\tau \,/\, C$ soit équivalent à $S(\ell)$, au sens de la partie 4.2.3. Voici les relations qu'on va utiliser :

$S \models v : \tau \,/\, C$	v, considérée dans un état mémoire de type S, est une valeur correcte du type non générique τ dans le contexte C
$S \models v : \sigma \,/\, C$	v, considérée dans un état mémoire de type S, est une valeur correcte du type générique σ dans le contexte C
$S \models e : E \,/\, C$	les valeurs contenues dans l'environnement d'évaluation e, considérées dans un état mémoire du type S, sont des valeurs correctes du type correspondant dans l'environnement de typage E
$\models s : S$	toutes les cases de l'état mémoire s sont des valeurs correctes des types correspondants dans S

Et en voici la définition exacte :

- $S \models cst : \texttt{unit} \,/\, C$ si cst est `()`
- $S \models cst : \texttt{int} \,/\, C$ si cst est un entier
- $S \models cst : \texttt{bool} \,/\, C$ si cst est `true` ou `false`
- $S \models (v_1, v_2) : \tau_1 \times \tau_2 \,/\, C$ si $S \models v_1 : \tau_1 \,/\, C$ et $S \models v_2 : \tau_2 \,/\, C$
- $S \models \ell : \tau_1$ `ref` $/\, C$ si $\ell \in \mathrm{Dom}(S)$ et $S(\ell) \equiv \tau_1 \,/\, C$.
- $S \models (f, x, a, e) : \tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2 \,/\, C$ s'il existe un environnement de typage E tel que

$$S \models e : E \,/\, C \quad \text{et} \quad E \vdash (f \ \texttt{where}\ f(x) = a) : \tau_1 \multimap\!\langle u \rangle\!\rightarrow \tau_2.$$

- $S \models v : \sigma \,/\, C$ si il n'y a pas de variables génériques dangereuses dans $\sigma \,/\, C$, et si $S \models v : \rho(\sigma) \,/\, \rho(C)$ pour tout renommage ρ des variables génériques de σ en variables non génériques.

- $S \models e : E \,/\, C$ si $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, et pour tout x dans $\mathrm{Dom}(E)$, on a $S \models e(x) : E(x) \,/\, C$.
- $\models s : S$ si $\mathrm{Dom}(s) = \mathrm{Dom}(S)$, et pour tout $\ell \in \mathrm{Dom}(s)$, on a $S \models s(\ell) : S(\ell)$.

Remarque. Comme dans la partie 3.3, on peut toujours, dans le cas des valeurs fonctionnelles, supposer $\mathrm{Dom}(E) = \mathcal{I}(f \texttt{ where } f(x) = a)$. Si ce n'est pas le cas, il suffit d'appliquer la règle de simplification pour restreindre E aux identificateurs libres dans $(f \texttt{ where } f(x) = a)$. □

On vérifie sans peine que les prédicats de typage sémantique définis ci-dessus sont stables par remplacement du typage de la mémoire S par un prolongement S' de S.

Proposition 4.11 *Soient v une valeur, S un type d'état mémoire, et $\tau \,/\, C$ et $\tau' \,/\, C'$ deux types non génériques contraints. Si $\tau \,/\, C \equiv \tau' \,/\, C'$, alors $S \models v : \tau \,/\, C$ équivaut à $S \models v : \tau' \,/\, C'$.*

Démonstration : vu la symétrie de $\equiv$, et le fait que $\tau = \tau'$, il suffit de montrer que $S \models v : \tau \,/\, C$ entraîne $S \models v : \tau \,/\, C'$. On procède par récurrence structurelle sur v.

• **Cas** $v = cst$**.** Trivialement vrai.

• **Cas** $v = (v_1, v_2)$**.** On a alors $\tau = \tau_1 \times \tau_2$, avec $S \models v_1 : \tau_1 \,/\, C$ et $S \models v_2 : \tau_2 \,/\, C$. Comme τ_1 et τ_2 sont sous-termes de τ, on a $\tau_1 \,/\, C \equiv \tau_1 \,/\, C'$, et de même pour τ_2. Par l'hypothèse de récurrence appliquée à v_1 et à v_2, il vient $S \models v_1 : \tau_1 \,/\, C'$ et $S \models v_2 : \tau_2 \,/\, C'$; d'où le résultat.

• **Cas** $v = \ell$**.** On a alors $\tau = \tau_1 \texttt{ ref}$, avec $\tau_1 \,/\, C \equiv S(\ell)$. Puisque τ_1 est un sous-terme de τ, on a $\tau_1 \,/\, C \equiv \tau_1 \,/\, C'$. Par transitivité de $\equiv$, il s'ensuit $\tau_1 \,/\, C' \equiv S(\ell)$, et donc $S \models \ell : \tau \,/\, C'$.

• **Cas** $v = (f, x, a, e)$**.** Alors $\tau = \tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2$, et on a un environnement de typage E tel que

$$S \models e : E \,/\, C \quad \text{et} \quad E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2 \,/\, C.$$

Soit $y \in \mathrm{Dom}(E)$. Par la remarque ci-dessus, on peut supposer y libre dans $(f \texttt{ where } f(x) = a)$. Vu la règle de typage des fonctions, la contrainte $E(y) \triangleleft u_n$ figure dans C. Il s'ensuit que $\mathcal{L}(E(y) \,/\, C) \subseteq \mathcal{L}(u_n \,/\, C) \subseteq \mathcal{L}(\tau \,/\, C)$. Donc, $E(y) \,/\, C \equiv E(y) \,/\, C'$, pour tout y dans le domaine de E. Ceci a deux conséquences. Premièrement, pour tout $y \in \mathrm{Dom}(E)$, on peut appliquer l'hypothèse de récurrence à $e(y)$ et $E(y)$; il vient $S \models e(y) : E(y) \,/\, C'$. D'où $S \models e : E \,/\, C'$. Deuxièmement, $E \,/\, C \equiv E \,/\, C'$. Appliquant la règle de simplification aux prémisses suivantes :

$$E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2 / C \qquad \tau_1 \multimap\!\langle u \rangle\!\rightarrow \tau_2 / C \equiv \tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2 / C' \qquad E / C \equiv E / C',$$

on déduit que

$$E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2 \,/\, C'.$$

Ceci achève d'établir le résultat annoncé : $S \models (f, x, a, e) : \tau_1 \multimap\!\langle u \rangle\!\rightarrow \tau_2 \,/\, C'$. □

On donne maintenant deux résultats de stabilité du typage sémantique par substitution de variables non dangereuses dans le type.

Proposition 4.12 *Soient v une valeur, τ un type non générique, C un ensemble de contraintes et S un type d'état mémoire tels que $S \models v : \tau \,/\, C$. Soit θ un renommage de variables non génériques en variables non génériques tel que* $\mathrm{Dom}(\theta) \cap \mathcal{D}(\tau \,/\, C) = \emptyset$. *Alors $S \models v : \theta(\tau) \,/\, \theta(C)$.*

Démonstration : on procède par récurrence structurelle sur v.

• **Cas** $v = cst$**.** Immédiat, puisque τ est alors un type de base, et donc $\theta(\tau) = \tau$.

• **Cas** $v = (v_1, v_2)$ **et** $\tau = \tau_1 \times \tau_2$**.** Comme $\mathcal{D}(\tau/C) = \mathcal{D}(\tau_1/C) \cup \mathcal{D}(\tau_2/C)$, on a $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau_1/C) = \emptyset$ et $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau_2 \,/\, C) = \emptyset$. Par hypothèse de récurrence, on a donc $S \models v_1 : \theta(\tau_1) \,/\, \theta(C)$ et $S \models v_2 : \theta(\tau_2) \,/\, \theta(C)$. D'où $S \models (v_1, v_2) : \theta(\tau_1 \times \tau_2) \,/\, \theta(C)$.

• **Cas** $v = \ell$ **et** $\tau = \tau_1$ `ref`**.** Par définition de $\models$, on a $S(\ell) \equiv \tau_1 \,/\, C$. Comme $\mathcal{D}(\tau_1 \texttt{ ref} \,/\, C) = \mathcal{L}(\tau_1 \,/\, C)$, on a $\mathrm{Dom}(\theta) \cap \mathcal{L}(\tau_1 \,/\, C) = \emptyset$. Donc $\theta(\tau_1) \,/\, \theta(C) \equiv \tau_1 \,/\, C$ par la proposition 4.6. Par transitivité de $\equiv$, il s'ensuit $S(\ell) \equiv \theta(\tau_1) \,/\, \theta(C)$, et donc $S \models \ell : \theta(\tau_1 \texttt{ ref}) \,/\, \theta(C)$, comme attendu.

• **Cas** $v = (f, x, a, e)$ **et** $\tau = \tau_1 \multimap\langle u_n \rangle\rightarrow \tau_2$**.** Soit E un environnement de typage tel que

$$S \models e : E \,/\, C \quad \text{et} \quad E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle u_n \rangle\rightarrow \tau_2 \,/\, C.$$

Par la proposition 4.8, on obtient :

$$\theta(E) \vdash (f \texttt{ where } f(x) = a) : \theta(\tau_1 \multimap\langle u_n \rangle\rightarrow \tau_2) \,/\, \theta(C).$$

Il reste à montrer que

$$S \models e : \theta(E) \,/\, \theta(C). \tag{1}$$

On fixe y dans le domaine de E. On peut supposer y libre dans f `where` $f(x) = a$. Montrons $S \models e(y) : \theta(E(y)) \,/\, \theta(C)$. Soit ρ un renommage des variables génériques de $\theta(E(y))$ en variables non génériques. On peut décomposer $\rho \circ \theta$ en $\theta' \circ \rho$, où ρ' est un renommage des variables génériques de $E(y)$ en variables non génériques, et θ' un renommage de variables non génériques en variables non génériques, avec de plus $\mathrm{Dom}(\rho') = \mathrm{Dom}(\rho)$ et $\mathrm{Dom}(\theta') \subseteq \mathrm{Dom}(\theta) \cup \mathrm{Im}(\rho')$. Il découle de l'hypothèse $S \models e(y) : E(y) \,/\, C$ et de la définition de $\models$ sur les schémas de types que $S \models e(y) : \rho'(E(y)) \,/\, \rho'(C)$. Comme de plus $E(y) \,/\, C$ ne contient pas de variables génériques dangereuses, on a $\mathrm{Dom}(\mathit{theta}') \cap \mathcal{D}(\rho'(E(y)) \,/\, \rho'(C)) = \emptyset$. Appliquant l'hypothèse de récurrence à $e(y)$, il vient $S \models e(y) : \theta'(\rho'(E(y))) \,/\, \theta'(\rho'(C))$, c'est-à-dire $S \models e(y) : \rho(\theta(E(y))) \,/\, \rho(\theta(C))$. Ce pour tout ρ, d'où $S \models e(y) : \theta(E(y)) \,/\, \theta(C)$. Il s'ensuit (1), et le résultat désiré. □

Proposition 4.13 *Soient v une valeur, $\tau \,/\, C$ un type non générique contraint et S un type d'état mémoire tels que $S \models v : \tau \,/\, C$. Soit φ :* `VarNongen` $\Rightarrow$ `TypNongen` $\cup$ `EtiqNongen` *une substitution telle que* $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau \,/\, C) = \emptyset$ *et $\varphi(C) \subseteq C$. Alors $S \models v : \varphi(\tau) \,/\, C$.*

Démonstration : on procède par récurrence structurelle sur v.

• **Cas** $v = cst$**.** Immédiat, puisque τ est alors un type de base, et donc $\varphi(\tau) = \tau$.

• **Cas** $v = (v_1, v_2)$ **et** $\tau = \tau_1 \times \tau_2$**.** Comme $\mathcal{D}(\tau/C) = \mathcal{D}(\tau_1/C) \cup \mathcal{D}(\tau_2/C)$, on a $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau_1/C) = \emptyset$ et $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau_2 \,/\, C) = \emptyset$. Par hypothèse de récurrence, on a donc $S \models v_1 : \varphi(\tau_1) \,/\, C$ et $S \models v_2 : \varphi(\tau_2) \,/\, C$. D'où $S \models (v_1, v_2) : \varphi(\tau_1 \times \tau_2) \,/\, C$.

• **Cas** $v = \ell$ **et** $\tau = \tau_1$ `ref`**.** Par définition de $\models$, on a $S(\ell) \equiv \tau_1 \,/\, C$. Comme $\mathcal{D}(\tau_1 \texttt{ ref} \,/\, C) = \mathcal{L}(\tau_1 \,/\, C)$, on a $\varphi(\tau_1) = \tau_1$. D'où $S(\ell) \equiv \varphi(\tau_1) \,/\, C$, et donc $S \models \ell : \varphi(\tau_1 \texttt{ ref}) \,/\, C$, comme attendu.

• **Cas** $v = (f, x, a, e)$ **et** $\tau = \tau_1 \multimap\!\langle u \rangle\!\rightarrow \tau_2$**.** Soit E un environnement de typage tel que

$$S \models e : E \,/\, C \quad \text{et} \quad E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\!\langle u \rangle\!\rightarrow \tau_2 \,/\, C.$$

Par la proposition 4.10, on obtient :

$$\varphi(E) \vdash (f \texttt{ where } f(x) = a) : \varphi(\tau_1 \multimap\!\langle u \rangle\!\rightarrow \tau_2) \,/\, C.$$

Il reste à montrer que

$$S \models e : \varphi(E) \,/\, C. \tag{1}$$

On fixe y dans $\mathrm{Dom}(E)$. Soit ρ un renommage des variables génériques de $\varphi(E(y))$ en variables non génériques. Il faut montrer que $S \models e(y) : \rho(\varphi(E(y))) \,/\, \rho(C)$. Soit ξ un renommage des variables génériques de $E(y)$ en des variables non génériques hors de portée de φ. Comme $S \models e(y) : E(y)/C$, on sait que $S \models e(y) : \xi(E(y)) \,/\, \xi(C)$. Comme $\mathrm{Im}(\xi)$ est hors de portée de φ, les deux substitutions φ et ξ commutent. Donc, $\varphi(\xi(C)) = \xi(\varphi(C)) \subseteq \xi(C)$. De plus,

$$\mathrm{Dom}(\varphi) \cap \mathcal{D}(\xi(E(y)) \,/\, \xi(C)) = \emptyset.$$

En effet, si α était dangereuse dans $\xi(E(y)) \,/\, \xi(C)$ et dans le domaine de $\mathrm{Dom}(\varphi)$, on aurait nécessairement $\xi(\alpha) = \alpha$ par l'hypothèse $\mathrm{Im}(\xi)$ hors de portée de φ, et donc α serait dangereuse dans $E(y) \,/\, C$ et dans le domaine de φ, contredisant l'hypothèse $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau \,/\, C) = \emptyset$. On peut donc appliquer l'hypothèse de récurrence à $e(y)$ de type $\xi(E(y)) \,/\, \xi(C)$, et à la substitution φ. Il vient

$$S \models e(y) : \varphi(\xi(E(y))) \,/\, \xi(C).$$

Soit alors $\theta = \rho \circ \xi^{-1}$. Comme les variables génériques libres dans $E(y)$ sont exactement les variables génériques libres dans $\varphi(E(y))$, la substitution θ est un renommage de variables non génériques en variables non génériques. De plus,

$$\mathrm{Dom}(\theta) \cap \mathcal{D}(\varphi(\xi(E(y))) \,/\, \xi(C)) = \emptyset,$$

puisque $\mathrm{Dom}(\theta)$, qui n'est autre que $\mathrm{Im}(\xi)$ car $\mathrm{Dom}(\rho) = \mathrm{Dom}(\xi)$, est hors de portée de φ, et puisque $\mathrm{Dom}(\theta) \cap \mathcal{D}(\xi(E(y)) \,/\, \xi(C)) = \emptyset$. Appliquant la proposition 4.12, il vient

$$S \models e(y) : \theta(\varphi(\xi(E(y)))) \,/\, \theta(\xi(C)).$$

Par construction, $\theta \circ \xi = \rho$. D'où $\theta \circ \varphi \circ \xi = \theta \circ \xi \circ \varphi = \rho \circ \varphi$. Le résultat fourni par la proposition 4.12 se lit donc comme :

$$S \models e(y) : \rho(\varphi(E(y))) \,/\, \rho(C).$$

Ceci vaut pour tout renommage ρ. D'où $S \models e(y) : \varphi(E(y)) \,/\, C$. Ce pour tout y. D'où (1), et le résultat annoncé. □

En corollaire des deux propriétés de stabilité par substitution, il vient que les opérations de généralisation et d'instanciation sont sémantiquement correctes.

Proposition 4.14 (Généralisation sémantique) *Soient S un type d'état mémoire, v_1 une valeur et $\tau_1 \,/\, C_1$ un type non générique contraint tels que $S \models v : \tau_1 \,/\, C_1$. Pour tout renommage ξ de variables non dangereuses dans $\tau_1 \,/\, C_1$ en variables génériques, on a $S \models v : \xi(\tau_1) \,/\, \xi(C_1)$. En corollaire, pour tout E, si $(\sigma, C) = \mathtt{Gen}(\tau_1, C_1, E)$, alors $S \models v : \sigma \,/\, C$.*

Démonstration : soit ρ un renommage des variables génériques de $\xi(\tau_1)$ en des variables non génériques. Il faut montrer que

$$S \models v : \rho(\xi(\tau_1)) \,/\, \rho(\xi(C_1)). \tag{1}$$

Posons $\theta = \rho \circ \xi$. Comme $\mathrm{Im}(\xi) = \mathrm{Dom}(\rho)$, la substitution θ est un renommage de variables non génériques en variables non génériques. De plus, $\mathrm{Dom}(\theta) \cap \mathcal{D}(\tau_1 \,/\, C_1) = \emptyset$, puisque $\mathrm{Dom}(\theta) = \mathrm{Dom}(\xi)$. Appliquant la proposition 4.12, il vient $S \models v : \theta(\tau_1) \,/\, \theta(C_1)$, c'est-à-dire (1). Ce pour tout renommage ρ, d'où $S \models v : \xi(\tau_1) \,/\, \xi(C_1)$.

Le corollaire est immédiat par définition de `Gen`. □

Proposition 4.15 (Instanciation sémantique) *Soient v une valeur, $\sigma \,/\, C$ un type générique contraint et S un type d'état mémoire tels que $S \models v : \sigma \,/\, C$. Alors $S \models v : \tau \,/\, C$ pour toute instance $\tau \leq \sigma \,/\, C$.*

Démonstration : soit $\tau \leq \sigma \,/\, C$. Soit $\theta : \mathcal{L}_g(\sigma) \Rightarrow \mathtt{TypNongen} \cup \mathtt{EtiqNongen}$ une substitution telle que $\tau = \theta(\sigma)$ et $\theta(C) \subseteq C$. Soit ρ un renommage des variables génériques de σ en des variables non génériques hors de portée de θ. Soit $\varphi = \theta \circ \rho^{-1}$. On a $\theta = \varphi \circ \rho$, et φ est une substitution de variables non génériques en des types non génériques. Par hypothèse $S \models v : \sigma \,/\, C$, on déduit que

$$S \models v : \rho(\sigma) \,/\, \rho(C).$$

Par construction de ρ, on a $\rho^{-1} \circ \varphi \circ \rho = \varphi \circ \rho$. Donc l'hypothèse $\theta(C) \subseteq C$, c'est-à-dire $\varphi(\rho(C)) \subseteq C$, entraîne $\varphi(\rho(C)) \subseteq \rho(C)$. De plus, les variables génériques de σ ne sont pas dangereuses dans $\sigma \,/\, C$. Comme ρ est un renommage, ceci entraîne que les variables de $\mathrm{Im}(\rho)$ ne sont pas dangereuses dans $\rho(\sigma) \,/\, \rho(C)$. D'où, compte tenu de $\mathrm{Dom}(\varphi) = \mathrm{Im}(\rho)$,

$$\mathrm{Dom}(\varphi) \cap \mathcal{L}(\rho(\sigma) \,/\, \rho(C)) = \emptyset.$$

Appliquant la proposition 4.13 à la substitution φ, il vient

$$S \models v : \varphi(\rho(\sigma)) \,/\, \rho(C).$$

Considérons le renommage ρ^{-1}. C'est un renommage de variables non génériques en variables génériques. De plus, les variables de $\mathrm{Dom}(\rho^{-1})$, c'est-à-dire de $\mathrm{Im}(\rho)$, ne sont pas dangereuses dans $\varphi(\rho(\sigma)) \,/\, \rho(\sigma)$. En effet, si $\alpha \in \mathrm{Im}(\rho)$ était dangereuse dans $\varphi(\rho(\sigma)) \,/\, \rho(C)$, elle serait aussi dangereuse dans $\rho(\sigma) \,/\, \rho(C)$, puisque α est hors de portée de φ ; mais c'est impossible, comme on l'a vu plus haut. Appliquant la proposition 4.14 au renommage ρ^{-1}, il vient

$$S \models v : \rho^{-1}(\varphi(\rho(\sigma))) \,/\, \rho^{-1}(\rho(C)),$$

c'est-à-dire $S \models v : \theta(\sigma) \,/\, C$, compte tenu de $\rho^{-1} \circ \varphi \circ \rho = \varphi \circ \rho = \theta$. C'est le résultat recherché. □

Une fois ces résultats préliminaires établis, passons à la propriété de sûreté forte.

Proposition 4.16 (Sûreté forte pour les références) *Soient a une expression, τ un type non générique, E un environnement de typage, C un ensemble de contraintes, e un environnement d'évaluation, s un état mémoire et S un type d'état mémoire tels que*

$$E \vdash a : \tau \,/\, C \quad et \quad S \models e : E\,|_{\mathcal{I}(a)} \,/\, C \quad et \quad \models s : S.$$

S'il existe une réponse r telle que $e \vdash a/s \Rightarrow r$, alors $r \neq \texttt{err}$; au contraire, r est de la forme v/s', et il existe un typage de la mémoire S' tel que :

$$S' \text{ prolonge } S \quad et \quad S' \models v : \tau \,/\, C \quad et \quad \models s' : S'.$$

Démonstration : la preuve procède par récurrence sur la taille de la dérivation d'évaluation, et, en cas d'égalité, sur la taille de la dérivation de typage. On raisonne par cas sur la dernière règle utilisée dans la dérivation de typage. La preuve est très proche de celle de la proposition 3.6. Je donne les cas qui ne sont pas entièrement semblables.

• **Règle d'instanciation d'une variable.**

$$\frac{\tau \leq E(x) \,/\, C}{E \vdash x : \tau \,/\, C}$$

Par hypothèse, $S \models e : E\,|_{\mathcal{I}(x)} \,/\, C$, donc $x \in \mathrm{Dom}(e)$ et $S \models e(x) : E(x) \,/\, C$. La seule évaluation possible est donc $e \vdash x/s \Rightarrow e(x)/s$. Par la proposition 4.15, on a bien $S \models e(x) : \tau \,/\, C$.

• **Règle des fonctions.**

$$\frac{\begin{array}{c} E + f \mapsto \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2 + x \mapsto \tau_1 \vdash a : \tau_2 \,/\, C \\ (E(y) \lhd u_n) \in C \text{ pour tout } y \text{ libre dans } f \ \texttt{where}\ f(x) = a \end{array}}{E \vdash (f \ \texttt{where}\ f(x) = a) : \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2 \,/\, C}$$

La seule évaluation possible est $e \vdash (f \ \texttt{where}\ f(x) = a)/s \Rightarrow (f, x, a, e)/s$. On a bien $S \models (f, x, a, e) : \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2 \,/\, C$ par définition de $\models$, en prenant $E\,|_{\mathcal{I}(a)}$ pour l'environnement de typage requis. $S' = S$ convient.

• **Règle de l'application.**

$$\frac{E \vdash a_1 : \tau_2 \multimap\langle u_n\rangle\rightarrow \tau_1 \,/\, C \qquad E \vdash a_2 : \tau_2 \,/\, C}{E \vdash a_1(a_2) : \tau_1 \,/\, C}$$

On a trois possibilités d'évaluation. La première conclut $r = \texttt{err}$ parce que $e \vdash a_1 \Rightarrow r_1$ et r_1 n'est pas de la forme $(f, x, a_0, e_0)/s'$; mais elle est exclue par l'hypothèse de récurrence appliquée à a_1, qui nous dit que $r_1 = v_1/s_1$ et $\models v_1 : \tau_2 \to \tau_1 \,/\, C$, et donc a fortiori v_1 est une fermeture. La

deuxième possibilité d'évaluation conclut $r = \texttt{err}$ parce que $e \vdash a_2 \Rightarrow \texttt{err}$; elle est de même exclue par l'hypothèse de récurrence appliquée à a_2. On est donc dans le troisième cas d'évaluation :

$$\frac{e \vdash a_1/s \Rightarrow (f, x, a_0, e_0)/s_1 \qquad e \vdash a_2/s_1 \Rightarrow v_2/s_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0/s_2 \Rightarrow r}{e \vdash a_1(a_2) \Rightarrow r}$$

Par hypothèse de récurrence appliquée à a_1, on obtient un typage mémoire S_1 tel que :

$$S_1 \models v_1 : \tau_2 \multimap\langle u_n\rangle\rightarrow \tau_1 \,/\, C \quad \text{et} \quad \models s_1 : S_1 \quad \text{et} \quad S_1 \text{ prolonge } S.$$

Par définition de $\models$, il existe donc E_0 tel que

$$S_1 \models e_0 : E_0 \quad \text{et} \quad E_0 \vdash (f \ \texttt{where}\ f(x) = a_0) : \tau_2 \multimap\langle u_n\rangle\rightarrow \tau_1 \,/\, C.$$

Une seule règle de typage permet de dériver le résultat de droite ; est donc vraie sa première prémisse :

$$E_0 + f \mapsto (\tau_2 \multimap\langle u_n\rangle\rightarrow \tau_1) + x \mapsto \tau_2 \vdash a_0 : \tau_1 \,/\, C.$$

Appliquant de même l'hypothèse de récurrence à a_2, on obtient S_2 tel que

$$S_2 \models v_2 : \tau_2 \,/\, C \quad \text{et} \quad \models s_2 : S_2 \quad \text{et} \quad S_2 \text{ prolonge } S_1.$$

On considère alors les environnements :

$$e_2 = e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \qquad\qquad E_2 = E_0 + f \mapsto (\tau_2 \multimap\langle u_n\rangle\rightarrow \tau_1) + x \mapsto \tau_2$$

Les variables libres dans a_0 sont x, f, et les variables libres dans $(f \ \texttt{where}\ f(x) = a_0)$. On a donc $S_2 \models e_2 : E_2 \,|\, _{\mathcal{I}(a_0)} \,/\, C$. On peut alors appliquer l'hypothèse de récurrence à l'expression a_0, dans les environnements e_2 et E_2, et l'état mémoire $s_2 : S_2$. Il vient $r = v/s'$. De plus, il existe S' tel que :

$$S' \models v : \tau_1 \quad \text{et} \quad \models s' : S' \quad \text{et} \quad S' \text{ prolonge } S_2.$$

C'est le résultat attendu, puisque a fortiori S' prolonge S.

• **Règle du `let`.**

$$\frac{E \vdash a_1 : \tau_1 \,/\, C_1 \qquad (\sigma, C) = \texttt{Gen}(\tau_1, C_1, E) \qquad E + x \mapsto \sigma \vdash a_2 : \tau_2 \,/\, C}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2 \,/\, C}$$

Montrons $S \models e : E \,|\, _{\mathcal{I}(a_1)} \,/\, C_1$. Par définition de `Gen`, on a $C = \theta(C_1)$ et $E = \theta(E)$ pour un certain renommage θ de variables non dangereuses dans $E \,/\, C_1$ en variables génériques. Soit y un identificateur libre dans a_1. Montrons $S \models e(y) : E(y) \,/\, C_1$. Soit ρ une substitution des variables génériques de $E(y)$ en variables non génériques. Il faut établir $S \models e(y) : \rho(E(y)) \,/\, \rho(C_1)$. La substitution $\rho \circ \theta^{-1}$ est un renommage des variables génériques de $E(y)$ en variables non génériques. De l'hypothèse $S \models e : E \,/\, C$ s'ensuit donc $S \models e(y) : \rho(\theta^{-1}(E(y))) \,/\, \rho(\theta^{-1}(C))$, c'est-à-dire $S \models e(y) : \rho(E(y)) \,/\, \rho(C_1)$. Ce pour tout ρ et pour tout y, d'où $S \models e : E \,|\, _{\mathcal{I}(a_1)} \,/\, C_1$ comme annoncé.

On a deux possibilités d'évaluation. La première correspond au cas $e \vdash a_1 \Rightarrow \texttt{err}$. Elle est exclue par l'hypothèse de récurrence appliquée à a_1 et à e de type $E \,/\, C_1$. L'évaluation est donc de la forme :

$$\frac{e \vdash a_1/s \Rightarrow v_1/s_1 \qquad e + x \mapsto v_1 \vdash a_2/s_1 \Rightarrow r}{e \vdash (\texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2)/s \Rightarrow r}$$

Par hypothèse de récurrence appliquée à a_1, on obtient S_1 tel que :

$$S_1 \models v_1 : \tau_1 \,/\, C_1 \quad \text{et} \quad \models s_1 : S_1 \quad \text{et} \quad S_1 \text{ prolonge } S.$$

Par la proposition 4.14, il vient $S_1 \models v_1 : \sigma \,/\, C$. Notons alors

$$e_1 = e + x \mapsto v_1 \qquad\qquad E_1 = E + x \mapsto \sigma.$$

Pour tout y libre dans a_2, ou bien $y = x$, ou bien y est libre dans a ; dans les deux cas on a $S_1 \models e_1(y) : E_1(y) \,/\, C$. Donc $S_1 \models e_1 : E_1 \,|_{\mathcal{I}(a_2)} \,/\, C$. On applique l'hypothèse de récurrence à a_2 considérée dans les environnements e_1 et E_1, et dans l'état mémoire $s_1 : S_1$. On obtient que r est de la forme v/s', et il existe S' tel que

$$S' \models v : \tau_2 \,/\, C \quad \text{et} \quad \models s' : S' \quad \text{et} \quad S' \text{ prolonge } S_1.$$

C'est bien le résultat attendu.

• **Règle des primitives, cas `ref`.**

$$\frac{\tau \multimap\langle u_n\rangle\rightarrowtail \tau\ \ \texttt{ref} \leq t_g \multimap\langle u_g\rangle\rightarrowtail t_g\ \ \texttt{ref} \,/\, C \qquad E \vdash a : \tau \,/\, C}{E \vdash \texttt{ref}(a) : \tau\ \ \texttt{ref} \,/\, C}$$

Il y a deux possibilités d'évaluation. L'une mène à $r = \texttt{err}$ parce que a s'évalue en `err` ; elle est exclue par hypothèse de récurrence appliquée à a. On est donc dans le deuxième cas d'évaluation de $\texttt{ref}(a)$, qui se termine par :

$$\frac{e \vdash a/s_0 \Rightarrow v/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash \texttt{ref}(a)/s_0 \Rightarrow \ell/(s_1 + \ell \mapsto v)}$$

Par l'hypothèse de récurrence appliquée à a, on obtient S_1 tel que

$$S_1 \models v : \tau \,/\, C \quad \text{et} \quad \models s_1 : S_1 \quad \text{et} \quad S_1 \text{ prolonge } S.$$

On définit :

$$s' = s_1 + \ell \mapsto v \quad \text{et} \quad S' = S_1 + \ell \mapsto (\tau \,/\, C).$$

Comme $\mathrm{Dom}(s_1) = \mathrm{Dom}(S_1)$, on a $\ell \notin \mathrm{Dom}(S_1)$. D'où S' prolonge S_1, et donc aussi S. On a donc $S' \models v : \tau \,/\, C$, c'est-à-dire $S' \models s'(\ell) : S'(\ell)$. Il s'ensuit $\models s' : S'$ et $S' \models \ell : \tau\ \ \texttt{ref} \,/\, C$, comme souhaité.

• **Règle des primitives, cas !.**

$$\frac{\tau\ \ \texttt{ref} \multimap\langle u_n\rangle\rightarrowtail \tau \leq t_g\ \ \texttt{ref} \multimap\langle u_g\rangle\rightarrowtail t_g \,/\, C \qquad E \vdash a : \tau\ \ \texttt{ref} \,/\, C}{E \vdash !(a) : \tau \,/\, C}$$

On a trois possibilités d'évaluation. L'une mène à `err` parce que a s'évalue en une réponse qui n'est pas de la forme ℓ/s'. Elle est exclue par hypothèse de récurrence appliquée à a. La seconde possibilité est de la forme :

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash\, !a/s_0 \Rightarrow \texttt{err}}$$

Par hypothèse de récurrence appliquée à a, on obtient S_1 tel que $S_1 \models \ell : \tau\ \texttt{ref}$ et $\models s_1 : S_1$. Ceci implique $\ell \in \mathrm{Dom}(s_1)$; contradiction. Reste donc la troisième possibilité :

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash\, !a/s_0 \Rightarrow s_1(\ell)/s_1}$$

Par l'hypothèse de récurrence appliquée à a, on obtient S_1 tel que

$$S_1 \models \ell : \tau\ \texttt{ref} \,/\, C \quad \text{et} \quad \models s_1 : S_1 \quad \text{et} \quad S_1 \text{ prolonge } S.$$

On a donc $S_1(\ell) \equiv \tau \,/\, C$ et $S_1 \models s_1(\ell) : S_1(\ell)$. D'où $S_1 \models s_1(\ell) : \tau \,/\, C$ par la proposition 4.11, et le résultat recherché, avec $S' = S_1$.

• **Règle des primitives, cas :=.**

$$\frac{\tau\ \texttt{ref} \times \tau \multimap\langle u_n\rangle\rightarrowtail \texttt{unit} \leq t_g\ \texttt{ref} \times t_g \multimap\langle u_g\rangle\rightarrowtail \texttt{unit} \,/\, C \qquad E \vdash a : \tau\ \texttt{ref} \times \tau \,/\, C}{E \vdash :=(a) : \texttt{unit}}$$

Comme pour la primitive !, la seule possibilité d'évaluation est :

$$\frac{e \vdash a/s_0 \Rightarrow (\ell, v)/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash :=(a)/s_0 \Rightarrow \texttt{()}/(s_1 + \ell \mapsto v)}$$

Par l'hypothèse de récurrence appliquée à a, on obtient S_1 tel que

$$S_1 \models (\ell, v) : \tau\ \texttt{ref} \times \tau \,/\, C \quad \text{et} \quad \models s_1 : S_1 \quad \text{et} \quad S_1 \text{ prolonge } S.$$

Posons $s' = s_1 + \ell \mapsto v$ et $S' = S_1$. Comme $S_1(\ell) \equiv \tau \,/\, C$, on a bien $S' \models s'(\ell) : S'(\ell)$ par la proposition 4.11. Et, trivialement, $S' \models \texttt{()} : \texttt{unit} \,/\, C$. D'où le résultat.

• **Règle de simplification.**

$$\frac{E \vdash a : \tau \,/\, C \qquad \tau \,/\, C \equiv \tau' \,/\, C' \qquad E \,|_{\,\mathcal{I}(a)} \,/\, C \equiv E' \,|_{\,\mathcal{I}(a)} \,/\, C'}{E' \vdash a : \tau' \,/\, C'}$$

Par la proposition 4.11, $S \models e : E' \,|_{\,\mathcal{I}(a)} \,/\, C'$ entraîne $S \models e : E \,|_{\,\mathcal{I}(a)} \,/\, C$. On applique l'hypothèse de récurrence avec la même dérivation d'évaluation, et avec la dérivation de $E \vdash a : \tau \,/\, C$ comme dérivation de typage. Il vient $r = v/s'$ et un typage de la mémoire S' tel que

$$S' \text{ prolonge } S \quad \text{et} \quad S' \models v : \tau \,/\, C \quad \text{et} \quad \models s' : S'.$$

Comme $\tau \,/\, C \equiv \tau' \,/\, C'$, on a aussi $S' \models v : \tau' \,/\, C'$ par la proposition 4.11. C'est le résultat attendu. □

4.3.2 Canaux de communication

La preuve de sûreté pour la sémantique avec canaux est essentiellement la même que celle de la partie 3.3.2, avec des lemmes techniques similaires à ceux de la partie 4.3.1. Je me contente de définir les prédicats de typage sémantique et d'ébaucher les étapes de la preuve. On se donne un typage des canaux Γ, qui associe un type non générique contraint à chaque identificateur de canal c.

$$\text{Typage des canaux}: \quad \Gamma \quad ::= \quad [c_1 \mapsto \tau_1 \,/\, C_1, \ldots, c_n \mapsto \tau_n \,/\, C_n]$$

Voici les prédicats de typage sémantique utilisés :

$\Gamma \models v : \tau \,/\, C$	v est une valeur correcte du type non générique τ dans le contexte C
$\Gamma \models v : \sigma \,/\, C$	v est une valeur correcte du type générique σ dans le contexte C
$\Gamma \models e : E \,/\, C$	les valeurs contenues dans l'environnement d'évaluation e appartiennent bien aux schémas correspondants dans $E \,/\, C$
$\models w :? \Gamma$	les événements de réception (de la forme $c \,?\, v$) contenus dans la séquence d'événements w respectent les types des canaux attribués par Γ
$\models w :! \Gamma$	les événements d'émission (de la forme $c \,!\, v$) contenus dans la séquence d'événements w respectent les types des canaux attribués par Γ

Voici leur définition exacte :

- $\Gamma \models cst : \texttt{unit} \,/\, C$ si cst est `()`
- $\Gamma \models cst : \texttt{int} \,/\, C$ si cst est un entier
- $\Gamma \models cst : \texttt{bool} \,/\, C$ si cst est `true` ou `false`
- $\Gamma \models (v_1, v_2) : \tau_1 \times \tau_2 \,/\, C$ si $\Gamma \models v_1 : \tau_1 \,/\, C$, et $\Gamma \models v_2 : \tau_2 \,/\, C$
- $\Gamma \models c : \tau_1 \ \texttt{chan} \,/\, C$ si $\Gamma(c) \equiv \tau_1 \,/\, C$
- $\Gamma \models (f, x, a, e) : \tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2 \,/\, C$ s'il existe un environnement de typage E tel que

$$\Gamma \models e : E \,/\, C \quad \text{et} \quad E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \multimap\!\langle u_n \rangle\!\rightarrow \tau_2.$$

- $\Gamma \models v : \sigma \,/\, C$ si il n'y a pas de variables génériques dangereuses dans $\sigma \,/\, C$, et si $\Gamma \models v : \rho(\sigma) \,/\, \rho(C)$ pour tout renommage ρ des variables génériques de σ en variables non génériques.
- $\Gamma \models e : E \,/\, C$ si $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, et pour tout x dans $\mathrm{Dom}(E)$, on a $\Gamma \models e(x) : E(x) \,/\, C$.
- $\models w :? \Gamma$ si $\Gamma \models v : \Gamma(c)$ pour tout événement de réception $c \,?\, v$ appartenant à la séquence w
- $\models w :! \Gamma$ si $\Gamma \models v : \Gamma(c)$ pour tout événement d'émission $c \,!\, v$ appartenant à la séquence w.

Les propositions qui suivent sont les lemmes essentiels de la preuve de sûreté.

Proposition 4.17 (Stabilité par équivalence) *Soient v une valeur, Γ un typage des canaux, et $\tau \,/\, C$ et $\tau' \,/\, C'$ deux types non génériques contraints. Si $\tau \,/\, C \equiv \tau' \,/\, C'$, alors $\Gamma \models v : \tau \,/\, C$ équivaut à $\Gamma \models v : \tau' \,/\, C'$.*

Démonstration : même preuve que la proposition 4.11. □

Proposition 4.18 (Généralisation sémantique) *Soient Γ un typage des canaux, v_1 une valeur et $\tau_1 \,/\, C_1$ un type non générique contraint tels que $\Gamma \models v : \tau_1 \,/\, C_1$. Si $(\sigma, C) = \mathtt{Gen}(\tau_1, C_1, E)$, alors $\Gamma \models v : \sigma \,/\, C$.*

Démonstration : même preuve que la proposition 4.14. On emploie un lemme de stabilité de $\models$ par renommage de variables non dangereuses analogue à la proposition 4.12. □

Proposition 4.19 (Instanciation sémantique) *Soient v une valeur, $\sigma \,/\, C$ un type générique contraint et Γ un typage des canaux tels que $\Gamma \models v : \sigma \,/\, C$. Alors $\Gamma \models v : \tau \,/\, C$ pour toute instance $\tau \leq \sigma \,/\, C$.*

Démonstration : même preuve que la proposition 4.15. On emploie un lemme de stabilité de $\models$ par substitution de variables non dangereuses analogue à la proposition 4.13. □

On passe maintenant à la propriété de sûreté forte. Comme dans la partie 3.3.2, on se donne un terme clos bien typé a_0, dont toutes les sous-expressions de la forme $\mathtt{newchan}(a)$ sont distinctes. On fixe une dérivation $\mathcal{T}$ de typage de a_0, et une dérivation $\mathcal{E}$ d'évaluation de a_0. On construit un typage des canaux Γ adapté à $\mathcal{T}$ et $\mathcal{E}$ comme expliqué dans la partie 3.3.2.

Proposition 4.20 (Sûreté forte pour les canaux) *Soit $e \vdash a \stackrel{w}{\Longrightarrow} r$ la conclusion d'une sous-dérivation de $\mathcal{E}$, et $E \vdash a : \tau \,/\, C$ la conclusion d'une sous-dérivation de $\mathcal{T}$, pour la même expression a. On suppose $\Gamma \models e : E \mid_{\mathcal{I}(a)} / C$.*

1. *Si $\models w :? \Gamma$, alors $r \neq \mathtt{err}$; au contraire r est une valeur v, qui vérifie $\Gamma \models v : \tau \,/\, C$, et de plus $\models w :! \Gamma$.*

2. *Si $w = w'.c\,!\,v.w''$ et $\models w' :? \Gamma$, alors $\Gamma \models v : \Gamma(c)$.*

Démonstration : la preuve est presque identique à celle de la proposition 3.9. On emploie le lemme 4.19 pour le cas des variables, le lemme 4.18 pour le cas du `let`, et le lemme 4.17 pour le cas des primitives ! et ?, et pour le cas de la règle de simplification. Les autres cas sont exactement les mêmes que dans la preuve de la proposition 3.9. □

4.3.3 Continuations

La preuve de sûreté du système indirect pour le calcul avec continuations reprend l'essentiel des parties 3.3.3 (pour la preuve de sûreté proprement dite) et 4.3.1 (pour les lemmes techniques). On emploie les relations de typage sémantique suivantes :

$\models v : \tau \,/\, C$	v est une valeur correcte du type non générique τ dans le contexte C
$\models v : \sigma \,/\, C$	v est une valeur correcte du type générique σ dans le contexte C
$\models e : E \,/\, C$	l'environnement d'évaluation e contient des valeurs qui appartiennent bien aux schémas correspondants dans $E \,/\, C$
$\models k :: \tau \,/\, C$	la continuation k accepte toutes les valeurs du type $\tau \,/\, C$

Voici leur définition exacte :

- $\models cst : \texttt{unit} \,/\, C$ si cst est `()`
- $\models cst : \texttt{int} \,/\, C$ si cst est un entier
- $\models cst : \texttt{bool} \,/\, C$ si cst est `true` ou `false`
- $\models (v_1, v_2) : \tau_1 \times \tau_2 \,/\, C$ si $\models v_1 : \tau_1 \,/\, C$, et $\models v_2 : \tau_2 \,/\, C$
- $\models k : \tau_1 \ \texttt{cont} \,/\, C$ si $\models k :: \tau_1 \,/\, C$
- $\models (f, x, a, e) : \tau_1 \multimap\langle u_n\rangle\rightarrowtail \tau_2 \,/\, C$ s'il existe un environnement de typage E tel que
$$\models e : E \,/\, C \quad \text{et} \quad E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \multimap\langle u_n\rangle\rightarrowtail \tau_2.$$
- $\models v : \sigma \,/ C$ si il n'y a pas de variables génériques dangereuses dans $\sigma \,/ C$, et si $\models v : \rho(\sigma) \,/\, \rho(C)$ pour tout renommage ρ des variables génériques de σ en variables non génériques.
- $\models e : E \,/\, C$ si $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, et pour tout x dans $\mathrm{Dom}(E)$, on a $\models e(x) : E(x) \,/\, C$.
- $\models \texttt{stop} :: \tau \,/\, C$ pour tout type $\tau \,/\, C$
- $\models \texttt{app1c}(a, e, k) :: \tau_1 \multimap\langle u_n\rangle\rightarrowtail \tau_2 \,/\, C$ s'il existe un environnement de typage E tel que
$$E \vdash a : \tau_1 \,/\, C \quad \text{et} \quad \models e : E \,/\, C \quad \text{et} \quad \models k :: \tau_2 \,/\, C$$
- $\models \texttt{app2c}(f, x, a, e, k) :: \tau \,/\, C$ s'il existe un environnement de typage E, un type τ' et une étiquette u_n tels que
$$E \vdash (f \ \texttt{where} \ f(x) = a) : \tau \multimap\langle u_n\rangle\rightarrowtail \tau' \,/\, C \quad \text{et} \quad \models e : E \,/\, C \quad \text{et} \quad \models k :: \tau' \,/\, C$$
- $\models \texttt{letc}(x, a, e, k) :: \tau \,/\, C$ s'il existe un environnement de typage E et un type τ' tels que
$$E + x \mapsto \texttt{Gen}(\tau, E) \vdash a : \tau' \,/\, C \quad \text{et} \quad \models e : E \,/\, C \quad \text{et} \quad \models k :: \tau' \,/\, C$$
- $\models \texttt{pair1c}(a, e, k) :: \tau \,/\, C$ s'il existe un environnement de typage E et un type τ' tels que
$$E \vdash a : \tau' \,/\, C \quad \text{et} \quad \models e : E \,/\, C \quad \text{et} \quad \models k :: \tau \times \tau' \,/\, C$$
- $\models \texttt{pair2c}(v, k) :: \tau \,/\, C$ s'il existe un type τ' tel que
$$\models v : \tau' \,/\, C \quad \text{et} \quad \models k :: \tau' \times \tau \,/\, C$$

- $\models \texttt{primc}(\texttt{callcc}, k) :: \tau\ \texttt{cont} \dashv\langle u_n\rangle\rightarrow \tau \,/\, C$ si $\models k :: \tau \,/\, C$
- $\models \texttt{primc}(\texttt{throw}, k) :: \tau\ \texttt{cont} \times \tau \,/\, C$ pour tout τ.

Les propositions qui suivent sont les lemmes essentiels de la preuve de sûreté.

Proposition 4.21 (Stabilité par équivalence) *Soient $\tau \,/\, C$ et $\tau' \,/\, C'$ deux types non génériques contraints tels que $\tau \,/\, C \equiv \tau' \,/\, C'$. Pour toute valeur v, $\models v : \tau \,/\, C$ équivaut à $\models v : \tau' \,/\, C'$. Pour toute continuation k, $\models k :: \tau \,/\, C$ équivaut à $\models k :: \tau' \,/\, C'$.*

Démonstration : la preuve est par récurrence structurelle sur v et k. Les cas sont semblables à ceux de la preuve de la proposition 4.11. □

Proposition 4.22 (Généralisation sémantique) *Soient v_1 une valeur et $\tau_1 \,/\, C_1$ un type non générique contraint tels que $\models v : \tau_1 \,/\, C_1$. Si $(\sigma, C) = \texttt{Gen}(\tau_1, C_1, E)$, alors $\models v : \sigma \,/\, C$.*

Démonstration : même preuve que la proposition 4.14. On emploie un lemme analogue à la proposition 4.12 : si $\models v : \tau \,/\, C$ et si θ est un renommage de variables non dangereuses dans $\tau \,/\, C$, alors $\models v : \theta(\tau) \,/\, \theta(C)$. Ce lemme se prouve comme le lemme 4.12. Le seul cas qui diffère est $v = k$. Dans ce cas, $\tau = \tau_1\ \texttt{cont}$ et $\mathcal{D}(\tau \,/\, C) = \mathcal{L}(\tau_1 \,/\, C)$. Donc $\tau_1 \,/\, C \equiv \theta(\tau_1) \,/\, \theta(C)$ par la proposition 4.6. Comme $\models k :: \tau_1 \,/\, C$, il découle de la proposition 4.21 que $\models k :: \theta(\tau_1) \,/\, \theta(C)$. D'où $\models k : \theta(\tau)\ \texttt{cont} \,/\, \theta(C)$, comme annoncé. □

Proposition 4.23 (Instanciation sémantique) *Soient v une valeur et $\sigma \,/\, C$ un type générique contraint tels que $\Gamma \models v : \sigma \,/\, C$. Alors $\models v : \tau \,/\, C$ pour toute instance $\tau \leq \sigma \,/\, C$.*

Démonstration : même preuve que la proposition 4.15. On emploie un lemme analogue à la proposition 4.13 : si $\models v : \tau \,/\, C$ et si φ est une substitution de variables non dangereuses par des types non génériques telle que $\varphi(C) \subseteq C$, alors $\models v : \varphi(\tau) \,/\, C$. Ce résultat se prouve exactement comme le lemme 4.13. □

Proposition 4.24 (Correction faible pour les continuations)

1. *Soient a une expression, τ un type non générique, C un ensemble de contraintes, E un environnement de typage, k une continuation et r une réponse tels que*

$$E \vdash a : \tau \,/\, C \quad et \quad \models e : E \,|_{\mathcal{I}(a)} \,/\, C \quad et \quad \models k :: \tau \,/\, C \quad et \quad e \vdash a; k \Rightarrow r.$$

   Alors $r \neq \texttt{err}$.

2. *Soient v une valeur, k une continuation, τ un type non générique, C un ensemble de contraintes et r une réponse tels que*

$$\models v : \tau \,/\, C \quad et \quad \models k :: \tau \,/\, C \quad et \quad \vdash v \triangleright k \Rightarrow r.$$

   Alors $r \neq \texttt{err}$.

Démonstration : la preuve décalque celle de la proposition 3.12. Le cas où a est une variable est réglé par le lemme 4.23 ; le cas où k est une continuation `letc`, par le lemme 4.22. Les autres cas sont exactement les mêmes que dans la preuve de la proposition 3.12. □

4.4 Inférence de types

Dans cette partie, on montre que tout programme bien typé dans le système de types indirect possède un type principal, et on fournit un algorithme pour calculer ce type.

4.4.1 Unification

L'algorithme de Damas-Milner s'adapte facilement au système de types indirect. En effet, l'algèbre des expressions de types possède la propriété d'unificateur principal.

Proposition 4.25 *Si deux types τ_1 et τ_2 possèdent un unificateur, alors ils possèdent un unificateur principal. On note $\mathtt{mgu}(\tau_1, \tau_2)$ un unificateur principal de τ_1 et τ_2, s'il existe.*

Démonstration : les expressions de types forment une algèbre libre à deux sortes (les types d'expressions et les étiquettes) ; d'où le résultat annoncé. De plus, $\mathtt{mgu}(\tau_1, \tau_2)$ s'obtient par un des algorithmes usuels d'unification entre termes d'une algèbre libre, comme, par exemple, l'algorithme de Robinson [84]. □

Comme dans la partie 1.5, on impose par la suite la condition supplémentaire suivante sur $\mathtt{mgu}(\tau_1, \tau_2)$: toute variable non libre dans τ_1 ni dans τ_2 est hors de portée de $\mathtt{mgu}(\tau_1, \tau_2)$. Cette condition a deux conséquences utiles. Tout d'abord, si τ_1 et τ_2 sont des types non génériques, la substitution $\mathtt{mgu}(\tau_1, \tau_2)$ est une substitution de variables de types non génériques par des types non génériques. D'autre part, l'unificateur principal ne "raccroche" pas de contraintes supplémentaires aux types unifiés. Plus précisément :

Proposition 4.26 *Soient τ_1, τ_2 deux types, T un ensemble de types, et C un ensemble de contraintes. Si μ est l'unificateur principal de τ_1 et τ_2, alors*

$$\mu(C \restriction (\{\tau_1, \tau_2\} \cup T)) \;=\; \mu(C) \restriction (\{\mu(\tau_1)\} \cup \mu(T)).$$

(Comme dans la partie 4.2.3, on a noté $C \restriction T$, où T est un ensemble de types, la composante connexe des types de T dans C, c'est-à-dire la restriction de l'ensemble de contraintes C aux étiquettes libres dans $\tau \,/\, C$ pour $\tau \in T$.)

Démonstration : l'inclusion $\subseteq$ découle de la proposition 4.5. Supposons cette inclusion stricte. Nécessairement, C contient alors une contrainte sur une étiquette u non libre dans aucun des types contraints $\sigma \,/\, C$ (σ étant soit τ_1, soit τ_2, soit un des types de T), mais telle que $\mu(u)$ est libre dans un des $\mu(\sigma) \,/\, \mu(C)$. Ceci n'est possible que si $\mu(u) \neq u$. Donc u est directement libre dans τ_1 ou dans τ_2, puisque toute variable non libre dans τ_1 ou dans τ_2 est hors de portée de μ. Mézalors u est libre dans $\tau_1 \,/\, C$ ou dans $\tau_2 \,/\, C$, d'où contradiction. □

4.4.2 L'algorithme d'inférence

L'algorithme d'inférence prend en entrée une expression a, un environnement de typage E, un ensemble de contraintes initiales C et un ensemble infini de "nouvelles" variables non génériques V ; il retourne un type non générique τ (le type le plus général pour a), une substitution φ (représentant

les instanciations qu'on a dû effectuer dans E), un ensemble de contraintes C', et un sous-ensemble V' de V.

On note $\texttt{Inst}(\sigma, C, V)$ une instance triviale du type générique $\sigma \,/\, C$: choisissant θ renommage des variables génériques de σ en des variables non génériques de V, on prend :

$$\texttt{Inst}(\sigma, C, V) = (\theta(\sigma),\ \theta(C) \cup C,\ V \setminus \mathrm{Im}(\theta)).$$

Algorithme 4.1 On prend $\texttt{Infer}(a, E, C, V)$ égal au quadruplet (τ, C', φ, V') défini par :

Si a est x :
  $(\tau, C', V') = \texttt{Inst}(E(x), C, V)$ et $\varphi = [\,]$
Si a est cst :
  $(\tau, C', V') = \texttt{Inst}(\texttt{TypCst}(cst), C, V)$ et $\varphi = [\,]$
Si a est f `where` $f(x) = a_1$:
  soient $t_1 \in V$ et $t_2 \in V$ deux variables de types non génériques
  et $u \in V$ une étiquette non générique
  soit $C_0 = C \cup \{E(y) \lhd u \mid y \in \mathcal{I}(a)\}$
  soit $(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E + f \mapsto (t_2 \multimap\!\langle u\rangle\!\rightarrow t_1) + x \mapsto t_2,\ C_0,\ V \setminus \{t_1, t_2, u\})$
  soit $\mu = \texttt{mgu}(\varphi_1(t_1), \tau_1)$
  alors $\tau = \mu(\varphi_1(t_2 \multimap\!\langle u\rangle\!\rightarrow t_1))$ et $\varphi = \mu \circ \varphi_1$ et $C' = \mu(C_1)$ et $V' = V_1$
Si a est $a_1(a_2)$:
  soit $(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, C, V)$
  soit $(\tau_2, C_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), C_1, V_1)$
  soit $t \in V$ une variable de type non générique et $u \in V$ une étiquette non générique
  soit $\mu = \texttt{mgu}(\varphi_2(\tau_1), \tau_2 \multimap\!\langle u\rangle\!\rightarrow t)$
  alors $\tau = \mu(t)$ et $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ et $C' = \mu(C_2)$ et $V' = V_2 \setminus \{t, u\}$
Si a est `let` $x = a_1$ `in` a_2 :
  soit $(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, C, V)$
  soit $(\sigma, C_0) = \texttt{Gen}(\tau_1, C_1, \varphi_1(E))$
  soit $(\tau_2, C_2, \varphi_2, V_2) = \texttt{Infer}(a_2,\ \varphi_1(E) + x \mapsto \sigma,\ C_0,\ V_1)$
  alors $\tau = \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $C' = C_2$ et $V' = V_2$
Si a est (a_1, a_2) :
  soit $(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, C, V)$
  soit $(\tau_2, C_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), C_1, V_1)$
  alors $\tau = \tau_1 \times \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $C' = C_2$ et $V' = V_2$
Si a est $op(a_1)$:
  soit $(\tau_0, C_0, V_0) = \texttt{Inst}(\texttt{TypOp}(\texttt{op}), C, V)$
  soit $(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, C_0, V_0)$
  soit t une variable non générique et u une étiquette non générique, prises dans V_1
  soit $\mu = \texttt{mgu}(\tau_1 \multimap\!\langle u\rangle\!\rightarrow t, \tau_0)$
  alors $\tau = \mu(t)$ et $\varphi = \mu \circ \varphi_1$ et $C' = \mu(C_1)$ et $V' = V_1 \setminus \{t, u\}$

On dit que $\texttt{Infer}(a, E, C, V)$ n'est pas défini si, en cours de calcul, aucun cas ne s'applique ; en particulier, si on applique `mgu` à deux types non unifiables.

Quelques remarques sur les résultats de l'algorithme. Si $(\tau, C', \varphi, V') = \texttt{Infer}(a, E, C, V)$, et si $\mathcal{L}(E) \cap V = \emptyset$, et si $\mathcal{L}(C) \cap V = \emptyset$, alors :

1. φ est une substitution des variables non génériques dans les types non génériques
2. $V' \subseteq V$
3. les variables de V' ne sont pas libres dans τ', ni dans C'
4. les variables de V' sont hors de portée de φ, et donc non libres dans $\varphi(E)$
5. $C' \supseteq \varphi(C)$
6. aucune variable générique libre dans $\varphi(E)$ n'est libre dans $C' \setminus \varphi(C)$.

Proposition 4.27 (Correction de l'algorithme d'inférence) *Soient a une expression, E un environnement de typage, C un ensemble de contraintes et V un ensemble de variables non génériques tel que $\mathcal{L}(E) \cap V = \emptyset$ et $\mathcal{L}(C) \cap V = \emptyset$. Si $(\tau, \varphi, C', V_1) = \texttt{Infer}(a, E, C, V)$ est défini, alors on peut dériver $\varphi(E) \vdash a : \tau \,/\, C_1$.*

Démonstration : la preuve est par récurrence structurelle sur a, et suit de très près la preuve de la proposition 1.8, manipulations d'ensembles de contraintes en plus. La preuve repose essentiellement sur la stabilité du typage par substitution (proposition 4.8) et par ajout de contraintes (proposition 4.9). Je donne un cas de base et deux cas de récursion ; les autres cas sont similaires.

• **Cas $a = x$.** On a $(\tau, C', V') = \texttt{Inst}(E(x), C, V)$. Par définition de `Inst`, on a $\tau \leq E(x) \,/\, C'$. On peut donc bien dériver $E \vdash x : \tau \,/\, C'$.

• **Cas $a = (f \texttt{ where } f(x) = a_1)$.** Avec les notations de l'algorithme, on obtient par hypothèse de récurrence une preuve de :

$$\varphi_1(E + f \mapsto (t_1 \multimap\langle u\rangle\rightarrowtail t_2) + x \mapsto t_1) \vdash a_1 : \tau_1 \,/\, C_1.$$

Par la proposition 4.8, notant $\varphi = \mu \circ \varphi_1$, il vient une preuve de :

$$\varphi(E) + f \mapsto (\varphi(t_1) \multimap\langle\varphi(u)\rangle\rightarrowtail \varphi(t_2)) + x \mapsto \varphi(t_1) \vdash a_1 : \mu(\tau_1) \,/\, \mu(C_1).$$

La substitution μ étant un unificateur de $\varphi_1(t_2)$ et de τ_1, on a $\varphi(t_2) = \mu(\varphi_1(t_2)) = \mu(\tau_1)$. On a donc prouvé que :

$$\varphi(E) + f \mapsto (\varphi(t_1) \multimap\langle\varphi(u)\rangle\rightarrowtail \varphi(t_2)) + x \mapsto \varphi(t_1) \vdash a_1 : \varphi(t_2) \,/\, \mu(C_1).$$

De plus, C_1 contient $\varphi_1(C_0)$ par la remarque (5), et donc a fortiori les contraintes $\varphi_1(E(y)) \lhd \varphi_1(u)$ pour tout y libre dans a. Donc, $\mu(C_1)$ contient les contraintes $\varphi(E(y)) \lhd \varphi(u)$ pour tout y libre dans a, et on peut appliquer la règle de typage des fonctions, obtenant

$$\varphi(E) \vdash (f \texttt{ where } f(x) = a_1) : \varphi(t_1 \multimap\langle u\rangle\rightarrowtail t_2) \,/\, \mu(C_1).$$

C'est le résultat recherché.

• **Cas** $a = a_1(a_2)$**.** On applique l'hypothèse de récurrence aux deux appels récursifs de `Infer`. Il vient des preuves de :

$$\varphi_1(E) \vdash a_1 : \tau_1 \,/\, C_1 \qquad \varphi_2(\varphi_1(E)) \vdash a_2 : \tau_2 \,/\, C_2.$$

On applique la substitution $\mu \circ \varphi_2$ au jugement de gauche, et μ au jugement de droite. Par la proposition 4.8, il vient des preuves de :

$$\mu(\varphi_2(\varphi_1(E))) \vdash a_1 : \mu(\varphi_2(\tau_1)) \,/\, \mu(\varphi_2(C_1)) \qquad \mu(\varphi_2(\varphi_1(E))) \vdash a_2 : \mu(\tau_2) \,/\, \mu(C_2).$$

On étend alors $\mu(\varphi_2(C_1))$ en $\mu(C_2)$. Par la remarque (5), on a $\mu(\varphi_2(C_1)) \subseteq \mu(C_2)$. Par la remarque (6), aucune des variables génériques libres dans $\mu(\varphi_2(\varphi_1(E)))$ n'est libre dans $\mu(C_2) \setminus \mu(\varphi_2(C_1))$. (La substitution μ n'introduit pas de nouvelles variables génériques.) D'où, par la proposition 4.9 :

$$\mu(\varphi_2(\varphi_1(E))) \vdash a_1 : \mu(\varphi_2(\tau_1)) \,/\, \mu(C_2).$$

De plus, μ est un unificateur de $\varphi_2(\tau_1)$ et $\tau_2 \dashv\langle u \rangle\!\rightarrow t$. Prenant $\tau = \mu(t)$ et $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ et $C' = \mu(C_2)$, comme dans l'algorithme, on a donc établi :

$$\varphi(E) \vdash a_1 : \mu(\tau_2) \dashv\langle \mu(u) \rangle\!\rightarrow \tau \,/\, C' \qquad \varphi(E) \vdash a_2 : \mu(\tau_2) \,/\, C'.$$

Le résultat attendu s'ensuit par la règle de l'application de fonction :

$$\varphi(E) \vdash a_1(a_2) : \tau \,/\, C'.$$

□

Proposition 4.28 (Complétude de l'algorithme d'inférence) *Soient a une expression, E un environnement de typage, C un ensemble de contraintes et V un ensemble de variables génériques contenant une infinité de variables de types et une infinité d'étiquettes, et tel que $\mathcal{L}(E) \cap V = \emptyset$ et $\mathcal{L}(C) \cap V = \emptyset$. S'il existe un type $\overline{\tau}$, une substitution $\overline{\varphi}$ de variables non génériques en types non génériques, et un ensemble de contraintes $\overline{C}$ tels que*

$$\overline{\varphi}(E) \vdash a : \overline{\tau} \,/\, \overline{C} \quad \textit{et} \quad \overline{\varphi}(C) \subseteq \overline{C} \quad \textit{et} \quad \mathcal{L}_g(\overline{\varphi}(E)) \cap \mathcal{L}_g(\overline{C} \setminus \overline{\varphi}(C)) = \emptyset,$$

alors $(\tau, C', \varphi, V') = \texttt{Infer}(a, E, C, V)$ est bien défini, et il existe une substitution ψ telle que

$$\overline{\tau} = \psi(\tau) \quad \textit{et} \quad \overline{C} \supseteq \psi(C' \restriction \{\tau, \varphi(E)\}) \quad \textit{et} \quad \overline{\varphi} = \psi \circ \varphi \textit{ hors de } V.$$

(C'est-à-dire, $\overline{\varphi}(\alpha_n) = \psi(\varphi(\alpha_n))$ pour toute variable non générique $\alpha_n \notin V$.)

Démonstration : la preuve est par récurrence sur la dérivation de $\overline{\varphi}(E) \vdash a : \overline{\tau}/\overline{C}$, et par cas sur la dernière règle utilisée. La preuve suit de très près celle de la proposition 1.9, avec des manipulations de contraintes en plus. Je donne trois cas représentatifs.

• **Règle d'instanciation.**

$$\frac{\tau \leq \overline{\varphi}(E(x)) \,/\, \overline{C}}{\overline{\varphi}(E) \vdash x : \overline{\tau} \,/\, \overline{C}}$$

$\texttt{Infer}(x, E, C, V)$ est défini, puisque $x \in \mathrm{Dom}(e)$, et renvoie $\tau = \theta(E(x))$ et $C' = \theta(C) \cup C$ et $\varphi = [\,]$ et $V' = V \setminus \mathrm{Im}(\varphi)$, pour un certain renommage $\theta : \mathcal{L}_g(E(x)) \Leftrightarrow V$. Par définition de la relation d'instanciation, on a $\tau' = \rho(\overline{\varphi}(E(x)))$ pour une certaine substitution $\rho : \mathcal{L}_g(\varphi(E(x))) \Rightarrow \texttt{TypNongen}$, avec de plus $\rho(\overline{C}) \subset \overline{C}$. Prenons donc $\psi = \rho \circ \overline{\varphi} \circ \theta^{-1}$. On a, premièrement :

$$\psi(\tau) = \rho(\overline{\varphi}(\theta^{-1}(\tau))) = \rho(\overline{\varphi}(E(x))) = \tau'.$$

Ensuite, pour toute variable $\alpha_n \notin V$, on a $\alpha \notin \mathrm{Dom}(\theta^{-1})$, et par conséquent $\psi(\alpha_n) = \rho(\overline{\varphi}(\alpha_n)) = \overline{\varphi}(\alpha_n)$, puisque $\overline{\varphi}(\alpha_n)$ est un type non générique. Montrons pour finir $\psi(C') \subseteq \overline{C}$. On a :

$$\psi(C') = \psi(C) \cup \psi(\theta(C)) = \rho(\overline{\varphi}(\theta^{-1}(C))) \cup \rho(\overline{\varphi}(C)) = \rho(\overline{\varphi}(C)) \cup \rho(\overline{\varphi}(C)),$$

puisque $V \cap \mathcal{L}(C) = \emptyset$ et $\mathrm{Dom}(\theta^{-1}) \subseteq V$. Or,

$$\rho(\overline{\varphi}(C)) \subseteq \rho(\overline{C}) \subseteq \overline{C}.$$

D'où le résultat annoncé.

• **Règle des fonctions.**

$$\frac{\begin{array}{c}\overline{\varphi}(E) + f \mapsto \overline{\tau}_2 \multimap\!\langle \overline{u} \rangle\!\rightarrow \overline{\tau}_1 + x \mapsto \overline{\tau}_2 \vdash a_1 : \overline{\tau}_1 \,/\, \overline{C} \\ (\overline{\varphi}(E(y)) \triangleleft \overline{u}) \in \overline{C} \text{ pour tout } y \in \mathcal{I}(f \texttt{ where } f(x) = a)\end{array}}{\overline{\varphi}(E) \vdash (f \texttt{ where } f(x) = a_1) : \overline{\tau}_2 \multimap\!\langle \overline{u} \rangle\!\rightarrow \overline{\tau}_1 \,/\, \overline{C}}$$

Soient t_1 et t_2 deux variables de types non génériques et u une étiquette non générique, prises toutes trois dans V. Considérons l'environnement

$$E_1 = E + f \mapsto (t_2 \multimap\!\langle u \rangle\!\rightarrow t_1) + x \mapsto t_2$$

et la substitution

$$\overline{\varphi}_1 = \overline{\varphi} + t_1 \mapsto \overline{\tau}_1 + t_2 \mapsto \overline{\tau}_2 + u \mapsto \overline{u}.$$

La première prémisse de la règle ci-dessus se lit aussi $\overline{\varphi}_1(E_1) \vdash a : \overline{\tau}_1 \,/\, \overline{C}$. On applique l'hypothèse de récurrence à cette prémisse. Il vient que

$$(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E_1, C, V \setminus \{t_1, t_2, u\})$$

est défini. De plus, on obtient une substitution ψ_1 telle que :

$$\overline{\tau}_1 = \psi_1(\tau_1) \quad \text{et} \quad \overline{C} \supseteq \psi_1(C_1 \restriction \{\tau_1, \varphi_1(E_1)\}) \quad \text{et} \quad \overline{\varphi}_1 = \psi_1 \circ \varphi_1 \text{ hors de } V \setminus \{t_1, t_2, u\}.$$

On a $\psi_1(\varphi_1(t_1)) = \overline{\varphi}_1(t_1) = \overline{\tau}_1 = \psi_1(\tau_1)$. Les types $\varphi_1(t_1)$ et τ_1 admettent donc ψ_1 comme unificateur. Par conséquent, $\mu = \texttt{mgu}(\varphi_1(t_1), \tau_1)$ est bien défini. Il s'ensuit que $\texttt{Infer}(E, a, C, V)$ est bien défini. De plus, on a $\psi_1 = \psi \circ \mu$ pour une certaine substitution ψ. L'algorithme prend $\varphi = \mu \circ \varphi_1$ et $\tau = \mu(\varphi_1(t_2 \multimap\!\langle u \rangle\!\rightarrow t_1))$. On a :

$$\psi(\tau) = \psi(\mu(\varphi_1(t_2 \multimap\!\langle u \rangle\!\rightarrow t_1))) = \psi_1(\varphi_1(t_2 \multimap\!\langle u \rangle\!\rightarrow t_1)) = \overline{\varphi}_1(t_2 \multimap\!\langle u \rangle\!\rightarrow t_1) = \overline{\tau}_2 \multimap\!\langle \overline{u} \rangle\!\rightarrow \overline{\tau}_1 = \overline{\tau}.$$

(En effet, t_1, t_2 et u ne sont pas dans $V \setminus \{t_1, t_2, u\}$, donc ces variables ont la même image par $\overline{\varphi}_1$ et par $\psi_1 \circ \varphi_1$.) De même, pour tout $\alpha_n \notin V'$, on a a fortiori $\alpha_n \notin V$ et $\alpha \notin \{t_1, t_2, u\}$, d'où

$$\psi(\varphi(\alpha_n)) = \psi(\mu(\varphi_1(\alpha_n))) = \psi_1(\varphi_1(\alpha_n)) = \overline{\varphi}_1(\alpha_n) = \overline{\varphi}(\alpha_n).$$

Montrons pour finir $\overline{C} \supseteq \psi(C' \restriction \{\tau, \varphi(E)\})$. Par la proposition 4.26, il vient

$$\mu(C_1 \restriction \{\tau_1, \varphi_1(E_1)\}) = \mu(C_1) \restriction \{\mu(\tau_1), \mu(\varphi_1(E_1))\}.$$

De plus, par construction de E_1,

$$\mu(C_1) \restriction \{\mu(\tau_1), \mu(\varphi_1(E_1))\} = \mu(C_1) \restriction \{\mu(\tau_1), \mu(\varphi_1(E))\} = \mu(C_1) \restriction \{\tau, \varphi(E)\}.$$

En effet, les variables de types libres dans $\mu(\varphi_1(E_1)) \,/\, \mu(C_1)$ et non libres dans $\mu(\varphi_1(E)) \,/\, \mu(C_1)$ sont contenues dans les variables libres dans $\mu(\varphi_1(t_2 \prec\!\langle u \rangle\!\rightarrow t_1)) \,/\, \mu(C_1)$, c'est-à-dire libres dans $\tau \,/\, \mu(C_1)$. Or, on sait par l'hypothèse de récurrence que

$$\overline{C} \supseteq \psi(\mu(C_1 \restriction \{\tau_1, \varphi_1(E_1)\})).$$

D'où

$$\overline{C} \supseteq \psi(\mu(C_1) \restriction \{\tau, \varphi(E)\}).$$

D'autre part, par la règle de typage du `where`, on sait que

$$(\overline{\varphi}(E(y)) \lhd \overline{u}) \in \overline{C} \text{ pour tout } y \in \mathcal{I}(f \texttt{ where } f(x) = a).$$

On a $\overline{\varphi}(E) = \psi(\varphi(E))$, puisque les variables libres de E sont hors de V. Donc

$$(\psi(\varphi(E(y))) \lhd \overline{u}) \in \overline{C} \text{ pour tout } y \in \mathcal{I}(f \texttt{ where } f(x) = a).$$

Comme, par définition, $C' = \mu(C_1) \cup \{\varphi(E(y)) \lhd \varphi(u) \mid y \in \mathcal{I}(a)\}$, on a donc bien établi

$$\overline{C} \supseteq \psi(C' \restriction \{\tau, \varphi(E)\}).$$

Ceci achève la preuve du cas des fonctions.

• **Règle du `let`.**

$$\frac{\overline{\varphi}(E) \vdash a_1 : \overline{\tau}_1 \,/\, \overline{C}_1 \qquad (\overline{\sigma}, \overline{C}) = \texttt{Gen}(\overline{\tau}_1, \overline{C}_1, \overline{\varphi}(E)) \qquad \overline{\varphi}(E) + x \mapsto \overline{\sigma} \vdash a_2 : \overline{\tau}_2 \,/\, \overline{C}}{\overline{\varphi}(E) \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \overline{\tau}_2 \,/\, \overline{C}}$$

On applique l'hypothèse de récurrence à la prémisse de gauche. On apprend que $(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, C, V)$ est défini, et on obtient une substitution ψ_1 telle que

$$\overline{\tau}_1 = \psi_1(\tau_1) \quad \text{et} \quad \overline{C}_1 \supseteq \psi_1(C_1 \restriction \{\tau_1, \varphi_1(E)\}) \quad \text{et} \quad \overline{\varphi} = \psi_1 \circ \varphi_1 \text{ hors de } V.$$

On a en particulier $\overline{\varphi}(E) = \psi_1(\varphi_1(E))$.

Soit $(\sigma, C_0) = \texttt{Gen}(\tau_1, C_1, \varphi_1(E))$. On vérifie que toute instance de $\overline{\sigma} \,/\, \overline{C}$ est aussi instance de $\psi_1(\sigma) \,/\, \overline{C}$. De la prémisse droite de la règle `let`, on peut donc déduire une preuve de

$$\psi_1(\varphi_1(E) + x \mapsto \sigma) \vdash a_2 : \overline{\tau}_2 \,/\, \overline{C}.$$

On applique l'hypothèse de récurrence à ce jugement. Il vient que

$$(\tau_2, C_2, \varphi_2, V_2) = \texttt{Infer}(a_2,\ \varphi_1(E) + x \mapsto \sigma,\ C_0,\ V_1)$$

est défini, et il existe une substitution ψ_2 telle que

$$\overline{\tau}_2 = \psi_2(\tau_2) \quad \text{et} \quad \overline{C} \supseteq \psi_2(C_2 \restriction \{\tau_2, \varphi_2(\varphi_1(E))\}) \quad \text{et} \quad \psi_1 = \psi_2 \circ \varphi_2 \text{ hors de } V_1.$$

On montre alors que $\psi = \psi_2$ convient, comme dans le cas des fonctions.

• **Règle de simplification.**

$$\frac{\overline{\varphi}(E) \vdash a : \overline{\tau} \,/\, \overline{C}' \qquad \overline{\tau} \,/\, \overline{C} \equiv \overline{\tau} \,/\, \overline{C}' \qquad \overline{\varphi}(E) \,/\, \overline{C} \equiv \overline{\varphi}(E) \,/\, \overline{C}'}{\overline{\varphi}(E) \vdash a : \overline{\tau} \,/\, \overline{C}}$$

On applique l'hypothèse de récurrence à la dérivation de $\overline{\varphi}(E) \vdash a : \overline{\tau}/\overline{C}'$. Il vient que $(\tau, \varphi, C', V') = \texttt{Infer}(a, E, C, V)$ est défini, et de plus, pour une certaine substitution ψ, on a

$$\overline{\tau} = \psi(\tau) \quad \text{et} \quad \overline{C}' \supseteq \psi(C \restriction \{\tau, \varphi(E)\}) \quad \text{et} \quad \overline{\varphi} = \psi \circ \varphi \text{ hors de } V.$$

Pour obtenir le résultat attendu, il suffit de montrer $\overline{C} \supseteq \psi(C \restriction \{\tau, \varphi(E)\})$. On a les inclusions suivantes :

$$\psi(C \restriction \{\tau, \varphi(E)\}) \subseteq \psi(C) \restriction \{\psi(\tau), \psi(\varphi(E))\} = \psi(C) \restriction \{\overline{\tau}, \overline{\varphi}(E)\} \subseteq \overline{C}' \restriction \{\overline{\tau}, \overline{\varphi}(E)\}.$$

D'après les prémisses 2 et 3 de la règle de simplification, l'ensemble de droite est égal à $\overline{C} \restriction \{\overline{\tau}, \overline{\varphi}(E)\}$, qui est inclus dans $\overline{C}$. D'où le résultat annoncé. □

4.5 Conservativité

Dans cette partie, on montre que le système de types indirect est une extension stricte du système de types de Milner (chapitre 1) : tout programme n'utilisant ni références, ni canaux, ni continuations et qui est bien typé dans le système de Milner est bien typé dans le système de types indirect.

Dans cette partie, on appelle "expression pure" toute expression a qui ne fait intervenir aucun des opérateurs sur les références, les canaux et les continuations introduits au chapitre 2. Une expression pure peut être typée de deux manières : ou bien dans le système de types de Milner (celui du langage purement applicatif du chapitre 1) ; ou bien dans le système de types indirect avec variables dangereuses et typage des fermetures (le système indirect, en abrégé) du présent chapitre. Le but de cette section est de montrer que toute expression pure bien typée dans le système de Milner est également bien typée dans le système indirect ; en d'autres termes, que le système indirect est une extension conservative du système de Milner.

Proposition 4.29 *Soit a une expression pure et ι un type de base. Si on peut prouver $[\,] \vdash a : \iota$ dans le système de Milner (partie 1.3.5), alors il existe C tel qu'on peut prouver $[\,] \vdash a : \iota \,/\, C$ dans le système indirect (partie 4.2.4).*

Pour prouver ce résultat, l'approche naturelle est de se donner une dérivation de typage dans le système de Milner et d'en déduire une dérivation dans le système indirect. Cette approche pose problème, car il faut annoter les types fonctionnels et synthétiser un ensemble de contraintes de manière globalement cohérente. Rien d'impossible ; mais cela duplique de gros morceaux de l'algorithme d'inférence de types pour le système indirect et de sa preuve de correction.

Remarque. La technique que je viens d'ébaucher marche en revanche fort bien pour montrer la réciproque du résultat de conservativité : tout programme pur bien typé dans le système indirect est bien typé dans le système de Milner. Il suffit en effet d'effacer les étiquettes et les contraintes de la dérivation de typage dans le système indirect, et de transformer les types génériques en schémas de types, pour obtenir une dérivation de typage valide dans le système de Milner. □

L'approche que je vais suivre, plus économique, consiste à comparer les algorithmes d'inférence de types pour les deux systèmes, et montrer que si l'algorithme pour le système de Milner termine avec succès, alors l'algorithme pour le système indirect termine avec succès. L'idée est que ces deux algorithmes effectuent des opérations presque identiques sur des données presque semblables, et que l'unification entre deux types indirects ne peut pas échouer à cause des étiquettes (qui sont de simples variables, et donc toujours unifiables entre elles.)

Pour exprimer précisément ce lien entre les deux algorithmes d'inférence, on définit une opération d'EFFACEMENT D'ÉTIQUETTES, notée $\Downarrow$, qui projette les expressions de types du système indirect sur des expressions de types du système de Milner. J'appelle SQUELETTE de σ le type $\sigma\Downarrow$.

$$\begin{aligned} \iota\Downarrow &= \iota \\ t\Downarrow &= t \\ (\sigma_1 \multimap\!\langle u\rangle\!\rightarrow \sigma_2)\Downarrow &= \sigma_1\Downarrow \rightarrow \sigma_2\Downarrow \\ (\sigma_1 \times \sigma_2)\Downarrow &= \sigma_1\Downarrow \times \sigma_2\Downarrow \end{aligned}$$

Par analogie, si V est un ensemble de variables de types et d'étiquettes, on note $V\Downarrow$ l'ensemble V privé de ses étiquettes. On définit une variante de $\Downarrow$, notée $\Downarrow_s$, qui transforme un type générique indirect en schéma de types du système de Milner :

$$\sigma\Downarrow_s = \forall t_1, \ldots, t_n.\, \sigma\Downarrow \quad \text{avec} \quad \{t_1 \ldots t_n\} = \mathcal{L}_g(\sigma)\Downarrow.$$

On étend $\Downarrow$ et $\Downarrow_s$ aux substitutions et aux environnements de typage, point à point. On vérifie facilement que $(\varphi \circ \psi)\Downarrow = \varphi\Downarrow \circ \psi\Downarrow$ pour toutes substitutions φ et ψ.

L'opérateur $\Downarrow$ n'est pas défini pour des types de la forme σ `ref`, σ `chan` ou σ `cont`. Toutes les expressions de types considérées par la suite ne contiennent aucun des constructeurs `ref`, `chan` ou `cont`. On se convainc facilement que l'algorithme d'inférence de types du système indirect ne produit pas de types contenant `ref`, `chan` ou `cont` lorsqu'il opère sur une expression pure.

La propriété du système indirect qui fait que tout marche est la suivante :

Proposition 4.30 *Deux types τ_1 et τ_2 sont unifiables si et seulement si leurs squelettes $\tau_1\Downarrow$ et $\tau_2\Downarrow$ sont unifiables. Dans ce cas, $\mathtt{mgu}(\tau_1, \tau_2)\Downarrow$ est égal, à un renommage près, à $\mathtt{mgu}(\tau_1\Downarrow, \tau_2\Downarrow)$.*

Démonstration : si φ est un unificateur de τ_1 et τ_2, on a immédiatement

$$\varphi\Downarrow(\tau_1\Downarrow) = (\varphi(\tau_1))\Downarrow = (\varphi(\tau_2))\Downarrow = \varphi\Downarrow(\tau_2\Downarrow)$$

et donc $\tau_1\Downarrow$ et $\tau_2\Downarrow$ sont unifiables, d'unificateur $\varphi\Downarrow$. Prenant pour φ l'unificateur principal de τ_1 et τ_2, il vient que $\mathtt{mgu}(\tau_1,\tau_2)\Downarrow$ est moins général que $\mathtt{mgu}(\tau_1\Downarrow,\tau_2\Downarrow)$.

Réciproquement, soit ψ un unificateur de $\tau_1\Downarrow$ et $\tau_2\Downarrow$. On fixe une étiquette u. On définit une opération $\Uparrow$ de relèvement des types de Milner en types indirects comme suit :

$$\begin{aligned}
\iota\Uparrow &= \iota \\
t\Uparrow &= t \\
(\tau_1 \to \tau_2)\Uparrow &= \tau_1\Uparrow \multimap\langle u\rangle\rightarrowtail \tau_2\Uparrow \\
(\tau_1 \times \tau_2)\Uparrow &= \tau_1\Uparrow \times \tau_2\Uparrow
\end{aligned}$$

On a évidemment $\tau\Uparrow\Downarrow = \tau$ pour tout type de Milner τ. Considérons la substitution φ définie par

$$\varphi = \psi\Uparrow + u_1 \mapsto u + \cdots + u_n \mapsto u,$$

où $u_1,\ldots,u_n$ sont les étiquettes libres dans τ_1 ou dans τ_2. On vérifie facilement que φ est un unificateur de τ_1 et de τ_2. De plus, $\varphi\Downarrow = \psi\Uparrow\Downarrow = \psi$. Prenons maintenant pour ψ l'unificateur principal de $\tau_1\Downarrow$ et $\tau_2\Downarrow$. La substitution φ associée est nécessairement de la forme $\xi\circ\mathtt{mgu}(\tau_1,\tau_2)$. D'où

$$\mathtt{mgu}(\tau_1\Downarrow,\tau_2\Downarrow) = \varphi\Downarrow = \xi\Downarrow\circ\mathtt{mgu}(\tau_1,\tau_2)\Downarrow.$$

Donc $\mathtt{mgu}(\tau_1\Downarrow,\tau_2\Downarrow)$ est moins général que $\mathtt{mgu}(\tau_1,\tau_2)\Downarrow$. D'où l'égalité à un renommage près annoncée. □

On va maintenant mettre en parallèle les deux algorithmes d'inférence, celui de Damas-Milner et celui pour le système indirect. Pour éviter toute confusion, on note $W(a,E,V)$ la fonction définie par l'algorithme de Damas-Milner (algorithme 1.1) et $I(a,E,C,V)$ la fonction définie par l'algorithme pour le système indirect (algorithme 4.1)

Proposition 4.31 *Soit a une expression pure, E un environnement de typage du système indirect, V un ensemble infini de variables et d'étiquettes, et C un ensemble de contraintes. Si $(\tau_w,\varphi_w,V_w) = W(a,E\Downarrow_s,V\Downarrow)$ est défini, alors $(\tau_i,C_i,\varphi_i,V_i) = I(a,E,C,V)$ est aussi défini, et on a*

$$\tau_i\Downarrow = \tau_w \quad et \quad \varphi_i\Downarrow = \varphi_w \quad et \quad V_i\Downarrow = V_w$$

à un renommage près des variables de V_i en variables de V_i.

Démonstration : par récurrence structurelle sur a. On donne trois cas représentatifs ; les autres cas sont similaires.

• **Cas $a = x$.** Comme $W(E\Downarrow_s,x)$ est défini, on a $x \in \mathrm{Dom}(E\Downarrow_s) = \mathrm{Dom}(E)$, et donc $I(x,E,C,V)$ est défini. On a de plus $(\tau_w,V_w) = \mathtt{Inst}(x,E(x)\Downarrow_s,V\Downarrow)$ et $(\tau_i,C_i,V_i) = \mathtt{Inst}(x,E(x)\Downarrow,V\Downarrow)$. Écrivant $\{t_1,\ldots,t_n,u_1,\ldots,u_m\} = \mathcal{L}_g(E(x))$, on a $E(x)\Downarrow_s = \forall t_1\ldots t_n.\,E(x)\Downarrow$. Donc τ_w est de la forme $[t_1 \mapsto t'_1,\ldots,t_n \mapsto t'_n](E(x)\Downarrow)$ pour certaines variables t'_i prises dans V. Choisissant

$\theta = [t_1 \mapsto t'_1, \ldots, t_n \mapsto t'_n, u_1 \mapsto u'_1, \ldots, u_m \mapsto u'_m]$ comme substitution d'instanciation de $E(x)$, on a donc bien

$$\tau_i\Downarrow = \theta(E(x))\Downarrow = \theta\Downarrow(E(x)\Downarrow) = \tau_w.$$

De même,

$$V_i\Downarrow = (V \setminus \{t'_1, \ldots, t'_n, u'_1, \ldots, u'_m\})\Downarrow = V\Downarrow \setminus \{t'_1, \ldots, t'_n\} = V_w.$$

Enfin, $\varphi_i = \varphi_w = [\,]$. D'où le résultat annoncé.

• **Cas** $a = (f \texttt{ where } f(x) = a_1)$**.** Soient t_1 et t_2 les nouvelles variables choisies par W. On peut choisir les mêmes dans I. On sait que

$$(\tau_{w1}, \varphi_{w1}, V_{w1}) = W(a_1, E\Downarrow_s + f \mapsto t_1 \to t_2 + x \mapsto t_1, V\Downarrow \setminus \{t_1, t_2\})$$

est défini. Appliquant l'hypothèse de récurrence à l'expression a_1, il s'ensuit que

$$(\tau_{i1}, C_{i1}, \varphi_{i1}, V_{i1}) = I(a_1, E + f \mapsto t_1 \multimap\langle u\rangle\rightarrowtail t_2 + x \mapsto t_1, C, V \setminus \{t_1, t_2, u\})$$

est bien défini. De plus, à un renommage près des variables de V en variables de V, on a $\tau_{i1}\Downarrow = \tau_{w1}$, et $\varphi_{i1}\Downarrow = \varphi_{w1}$, et $V_{i1}\Downarrow = V_{w1}$. On sait de plus que $\varphi_{w1}(t_2)$ et τ_{w1} sont unifiables. Or, $\varphi_{w1}(t_2) = (\varphi_{i1}(t_2))\Downarrow$ et $\tau_{w1} = \tau_{i1}\Downarrow$. Donc, par la proposition 4.30, $\varphi_{i1}(t_2)$ et τ_{i1} sont unifiables, ce qui établit que $I(a, E, C, V)$ est bien défini. De plus, notant $\mu_w = \texttt{mgu}(\varphi_{1w}(t_2), \tau_{w1})$ et $\mu_i = \texttt{mgu}(\varphi_{i1}(t_2), \tau_{i1})$, on a $\mu_w = \mu_i\Downarrow$. D'où, compte tenu de la définition de τ_w et de τ_i :

$$\tau_i\Downarrow = \mu_i(\varphi_{i1}(t_1 \multimap\langle u\rangle\rightarrowtail t_2))\Downarrow = \mu_i\Downarrow(\varphi_{i1}\Downarrow((t_1 \multimap\langle u\rangle\rightarrowtail t_2)\Downarrow)) = \mu_w(\varphi_{w1}(t_1 \to t_2)) = \tau_w.$$

De même,

$$\varphi_i\Downarrow = (\mu_i \circ \varphi_{i1})\Downarrow = \mu_i\Downarrow \circ \varphi_{i1}\Downarrow = \mu_w \circ \varphi_{w1} = \varphi_w$$
$$V_i\Downarrow = (V_{i1} \cup \{t_1, t_2, u\})\Downarrow = V_{w1} \cup \{t_1, t_2\} = V_w.$$

• **Cas** $a = (\texttt{let } x = a_1 \texttt{ in } a_2)$**.** On sait que $(\tau_{w1}, \varphi_{w1}, V_{w1}) = W(a_1, E\Downarrow_s, V\Downarrow)$ est défini. Donc, par hypothèse de récurrence, $(\tau_{i1}, \varphi_{i1}, C_1, V_{i1}) = I(a_1, E, C, V)$ est défini, et $\tau_{i1}\Downarrow = \tau_{w1}$, et $\varphi_{i1}\Downarrow = \varphi_{w1}$. Notons $(\sigma_i, C_i) = \texttt{Gen}(\tau_{i1}, \varphi_{i1}(E), C_{i1})$, et Notons $\sigma_w = \texttt{Gen}(\tau_{w1}, \varphi_{w1}(E\Downarrow_s))$. On va montrer que $\sigma' = \sigma\Downarrow_s$. Tout d'abord, $\mathcal{L}(\tau_{w1}) = \mathcal{L}(\tau_{i1}\Downarrow) = \mathcal{L}(\tau_{i1})\Downarrow$, et de même $\mathcal{L}(\varphi_{i1}(E\Downarrow_s)) = \mathcal{L}_n(\varphi_{i1}(E))\Downarrow$. Ensuite, $\mathcal{D}(\tau_{i1}/C_1) = \emptyset$. En effet, a_1 est un terme pur, donc le type contraint τ_{i1}/C_1 produit par I ne contient pas de sous-termes de la forme τ `ref`, ou τ `chan`, ou τ `cont`. De même, $\mathcal{D}(\varphi_{i1}(E)\,/\,C_1) = \emptyset$. Donc, les variables généralisées par I sont 1- les variables de types d'expressions généralisées par w, 2- des étiquettes, et 3- des variables de types d'expression indirectement libres dans τ_{i1} / C_{i1}, mais pas directement libres dans τ_{w1}. D'où, compte tenu des axiomes de la partie 1.3.3 sur les schémas, $\sigma_w = \sigma_i\Downarrow_s$. Comme par hypothèse $(\tau_{w2}, \varphi_{w2}, V_{w2}) = W(a_2, E\Downarrow_s + x \mapsto \sigma_w, V_1\Downarrow)$ est défini, il vient par hypothèse de récurrence que $(\tau_{i2}, \varphi_{i2}, C_2, V_{i2}) = I(a_2, E + x \mapsto \sigma_i, C_i, V_{i1})$ est défini. De plus, avec de plus $\tau_{i2}\Downarrow = \tau_{w2}$ et $\varphi_{i2}\Downarrow = \varphi_{i2}$ et $V_{i2}\Downarrow = V_{w2}$. On conclut donc que $I(a, E, C, V)$ est défini, et de plus :

$$\tau_i\Downarrow = \tau_{i2}\Downarrow = \tau_{w2} = \tau_w$$
$$\varphi_i\Downarrow = (\varphi_{i2} \circ \varphi_{i1})\Downarrow = \varphi_{i2}\Downarrow \circ \varphi_{i1}\Downarrow = \varphi_{w2} \circ \varphi_{w1} = \varphi_w$$
$$V_i\Downarrow = V_{i2}\Downarrow = V_{w2} = V_w.$$

C'est le résultat attendu. □

La proposition 4.29 s'ensuit de manière immédiate :

Démonstration : de la proposition 4.29. On rappelle les hypothèses. Soit a une expression pure et ι un type de base. On suppose qu'on peut prouver $[\,] \vdash a : \iota$ dans le système de Milner. Il découle de la complétude de l'algorithme W (proposition 1.9) que $W([\,], a)$ est défini, et renvoie un résultat de la forme (ι, φ'). Par la proposition 4.31, on déduit que $(\tau, \varphi, C) = I([\,], a)$ est défini. De plus $\tau\Downarrow = \iota$. La seule possibilité est $\tau = \iota$. Et puisque l'algorithme I est correct (proposition 4.27), on peut donc dériver $[\,] \vdash a : \iota \,/\, C$ dans le système indirect. C'est le résultat attendu. □

Remarque. Par une technique similaire, on peut montrer que le système de types indirects du présent chapitre est une extension du système du chapitre 3 : tout programme (pur ou non) bien typé dans le système du chapitre 3 est également bien typé dans le système indirect. □

Chapitre 5

Comparaisons

Dans ce chapitre, on compare les systèmes de types étudiés aux chapitres 3 et 4 avec d'autres systèmes de types polymorphes pour ML plus références proposés par ailleurs. (Certains de ces systèmes ont été appliqués à d'autres extensions de ML, en plus des références. La comparaison ne porte que sur les références, seul trait commun à tous ces systèmes.) Les critères de comparaison sont l'expressivité (combien de programmes corrects sont reconnus bien typés ?), d'une part, et de l'autre la facilité d'emploi de ces systèmes, en particulier dans le cadre de la programmation modulaire.

Pour ce qui est de l'expressivité, il se trouve qu'il n'y a pas en général d'inclusion stricte entre ces systèmes : presque toujours, il y a un exemple qui est bien typé dans l'un mais pas dans l'autre. Aussi, les seules comparaisons que l'on peut faire sont d'ordre pratique : voir comment sont typés des exemples représentatifs de situations réelles. C'est pourquoi je commence par présenter les programmes de test employés, avant de commenter les résultats obtenus avec les divers systèmes de types.

5.1 Présentation des programmes de test

La première série de tests mesure la capacité des systèmes considérés à typer de façon générique des fonctions qui opèrent sur des structures de données modifiables. Il s'agit du typage de la fonction

$$\texttt{let make\_ref} = \lambda \texttt{x}.\, \texttt{ref}(\texttt{x})$$

La fonction `make_ref` est le paradigme des fonctions génériques sur les tableaux, les matrices, les tables de hachage, les listes doublement chaînées, les *skip lists*, les *B-trees* et autres arbres équilibrés en place. Un bon système doit impérativement donner un type générique à la fonction `make_ref`. Un échec, ici, signifie qu'on ne peut pas avoir de bibliothèques implémentant une fois pour toutes ces structures de données utiles, mais complexes.

Le deuxième jeu de test dit s'il est possible ou non d'écrire des fonctions génériques dans un style non purement applicatif, par exemple en utilisant des références locales pour accumuler des résultats intermédiaires. Il s'agit de typer la fonction

```
let imp_map = λf. λl.
    let arg = ref(l) and res = ref([]) in
        while not null(!arg) do
            res := f(head(!arg)) :: !res;
            arg := tail(!arg)
        done;
        reverse(!res)
```

puis son application à la fonction identité et à liste vide, `imp_map id []`. On compare également le type de `imp_map` avec le type d'une fonction équivalente, mais purement applicative :

```
let rec appl_map = λf. λarg.
    if null(arg) then [] else
        f(head(arg)) :: appl_map f (tail(arg))
```

De nombreuses fonctions sur les graphes, par exemple (parcours en profondeur ou en largeur, calcul des composantes connexes, calcul du plus court chemin, tri topologique, etc [87, chapitres 29–34]) se typent de la même manière que la fonction `imp_map`. Un bon système doit donner le même type aux deux fonctions `imp_map` et `appl_map` ; l'application `imp_map id []` doit avoir le type polymorphe $\forall\alpha.\,\alpha$ `list`. Si un système ne passe pas ce test, cela indique que le polymorphisme n'y interagit pas de manière satisfaisante avec les traits non purement applicatifs.

La troisième série de test mesure la compatibilité entre les fonctions sur structures modifiables, d'une part, et fonctions d'ordre supérieur d'autre part. Elle consiste à appliquer la fonction identité à la fonction `make_ref` définie plus haut :

```
id (make_ref)
```

Une variante plus réaliste de ce test consiste à appliquer partiellement la fonctionnelle `appl_map` définie plus haut à la fonction `make_ref`. Enfin, on teste aussi l'application partielle de `imp_map`, cette fois, à la fonction identité. Un bon système doit donner le même type à `id make_ref` qu'à `make_ref`, et doit donner un type générique à `map make_ref` et à `imp_map id`. Le cas contraire est l'indice d'un mauvais traitement de la pleine fonctionnalité.

Les tests de la quatrième série sont plus artificiels et plus spécifiques à l'approche suivie dans cette Thèse. Ils détectent les problèmes mentionnés au début du chapitre 4 : apparition de types infinis (partie 4.1.1) et capture de variables par l'environnement (partie 4.1.2). On donne pour chaque problème deux versions du test, l'une sans références, l'autre avec références. Pour les types infinis, on considère :

```
let eta = λf. either f (λx. f(x))
```

où la fonction `either` est définie comme :

```
let either = λa. λb. if cond then a else b
```

où *cond* est une expression booléenne quelconque. La fonction `either` a pour but principal de forcer ses deux arguments à avoir le même type. Voici la variante avec références de ce test :

```
let eta_ref = λf.
    let r = ref(f) in
        either f (λx.!(either r (ref(f)))(x))
```

Le programme qui teste la capture de variables est le contre-exemple de la partie 4.1.2 :

```
let capt_id = λf.
    let id = λy. (either f (λz. y; z)); y
    in id(id)
```

Voici la variante avec quelques références en plus :

```
let capt_id_ref = λf.
    let id = λy.
        let r = ref(y) in
            either f (λz. either r (ref(y)); z);
        y
    in id(id)
```

Un bon système se doit d'être une extension stricte de ML, et donc de typer les versions "pures" de ces deux programmes. Les versions avec références sont sémantiquement correctes, mais on n'en voudra pas trop aux systèmes de types qui ne parviennent pas à détecter ce fait.

Le dernier test est lui aussi fort artificiel. Il s'agit d'imiter exactement, sur les types, la création de références, sans pour autant créer de référence. Voici le programme :

```
let fake_ref = (raise An_exception : α ref)
```

On a utilisé une exception et une contrainte de type pour plus de clarté, mais ces deux traits ne sont pas essentiels. Voici un exemple similaire qui ne les utilise pas :

```
let fake_ref = (λx.!x; x)(loop(0) where loop(x) = loop(x))
```

Ce test illustre la différence entre détecter la création des références, par une analyse supplémentaire ajoutée au typage, et détecter la présence des références, par simple examen des types : la première approche conclut correctement que rien de dangereux ne se produit ; la seconde voit que le résultat a le type α `ref`, et conclut incorrectement qu'on peut créer là une référence polymorphe.

5.2 Comparaison avec d'autres systèmes

Les résultats des tests sont résumés figure 5.1. On suit la convention que tous les exemples sont des phrases "toplevel", et que donc leur type doit être clos. Un tiret — signifie que l'exemple est rejeté parce que certaines variables ne sont pas généralisables dans son type, et donc son type ne peut pas être clos. Un smiley ☺ signifie que le type peut être clos, et donc que l'exemple est accepté.

Critère	CAML [5.2.1.1]	SML [5.2.1.2]	Damas [5.2.2.3]	SML-NJ [5.2.1.3]	Effets simples [5.2.2.1]	Effets d'alloc. [5.2.2.2]	Effets + régions [5.2.2.4]	Chap. 3	Chap. 4
`make_ref`	—	:)	:)	:)	:)	:)	:)	:)	:)
`imp_map`	—	:)	:)	:)	:)	:)	:)	:)	:)
`imp_map id []`	—	—	—	—	—	—	:)	:)	:)
Même type pour `imp_map` et `appl_map id(make_ref)`	—	—	—	—	—	—	:)	:)	:)
`appl_map(make_ref)`	—	—	—	—	:)	:)	:)	:)	:)
`imp_map(id)`	—	—	—	:)	:)	:)	:)	:)	:)
`eta`	:)	:)	:)	:)	:)	:)	:)	—	—
`eta_ref`	—	:)	?	:)	:)	—	—	—	—
`capt_id`	:)	:)	:)	:)	:)	:)	:)	—	—
`capt_id_ref`	—	:)	?	:)	:)	—	—	—	—
`fake_ref`	:)	:)	:)	:)	:)	:)	:)	—	—

Figure 5.1: Comparaison entre les différents systèmes

5.2.1 Les variables faibles

Une première approche pour contrôler les références polymorphes consiste à typer spécialement les primitives de création de références, de façon à marquer les variables de types qui sont libres dans le type de la référence créée. Ces variables marquées s'appellent variables faibles, ou variables impératives, et sont soumises à des conditions de généralisation plus restrictives que les variables usuelles. C'est cette approche qui est adoptée dans les trois premiers systèmes présentés ici, avec des restrictions plus ou moins fortes sur la généralisation.

5.2.1.1 Caml

La restriction la plus simple consiste à ne jamais généraliser une variable faible. Dès lors, une référence ne peut jamais avoir un type polymorphe, puisque 1- son type au moment de sa création ne comporte que des variables faibles, 2- le type de tout objet avec lequel la référence entre en contact va être affaibli de même (par unification), et 3- ces variables faibles ne sont jamais généralisées par la suite.

Historique. Cette idée semble être venue indépendamment à plusieurs personnes lorsque les références ont remplacé les variables modifiables de l'ancien LCF-ML [31, page 52]. Personne n'en a publié une présentation informelle, ni des règles de typage, ni une preuve de sûreté. Cette approche est toujours employée dans le système Caml, pour le typage des objets mutables [19, page 41] [98, page 79].

Résultats. Cette approche se révèle en pratique extrêmement restrictive. Les structures de données classiques (tables de hachage, etc.) ne peuvent pas être fournies par des bibliothèques génériques : le système ne tolère que des versions complètement monomorphes des opérations de création de ces structures. A fortiori, les fonctions génériques ne peuvent pas, en général, allouer des structures mutables contenant des résultat intermédiaires ; d'où une complète incompatibilité entre généricité et programmation impérative.

5.2.1.2 Standard ML

L'étape suivante consiste à permettre la généralisation des variables faibles dans le type d'une expression non-expansive : une expression dont l'évaluation ne crée pas de nouvelles références. Tofte a proposé l'approximation syntaxique suivante : est non-expansive une expression réduite à une variable, à une constante, ou à une fonction $\lambda x.\, a$; toutes les autres expressions sont supposées expansives. Considérons l'expression $\lambda \mathtt{x}.\, \mathtt{ref(x)}$, qui a le type faible $\alpha^* \to \alpha^*$ `list`. (Je note avec une étoile en exposant les variables faibles.) La variable α est marquée faible, parce que cette fonction crée une référence avec le type α. L'expression $\lambda \mathtt{x}.\, \mathtt{ref(x)}$ est non-expansive, dans la classification de Tofte. On peut donc généraliser α^* dans la suite du programme. Ainsi, l'exemple suivant est bien typé :

```
let make_ref = λx.ref(x) in
    ... make_ref(1) ... make_ref(true) ...
```

Le corps du `let` est en effet typé sous l'hypothèse `make_ref` : $\forall \alpha^*.\, \alpha^* \to \alpha^*$ `list`. Dans un tel schéma de type, la variable faible générique α^* peut être instanciée uniquement par des types

faibles, c'est-à-dire des types dont toutes les variables sont faibles elles aussi. C'est ce qui assure la sûreté de ce typage :

```
let make_ref = λx.ref(x) in
let r = make_ref(λx.x) in ...
```

L'application `make_ref`$(\lambda$`x.x`$)$ a le type $(\beta^* \to \beta^*)$ `ref`. La variable β est nécessairement faible, puisque le type par lequel α^* est instancié doit être faible. D'autre part, cette expression est classée expansive, puisque c'est une application et non une constante, une variable ou une fonction. Donc β^* n'est pas généralisée, empêchant toute utilisation incohérente de `r`.

Historique. Ce système a été introduit par Tofte en 1987, puis adopté dans le Standard ML. On le retrouve donc dans toutes les implémentations de ML qui suivent le Standard (Edinburgh ML, Poly ML, Poplog ML). Il est décrit avec précision dans la thèse de Tofte [91, 92] et dans la définition de Standard ML [64, 63]. Tofte a prouvé que ce système est sûr pour ML plus références [91, 92], en utilisant une sémantique relationnelle et des prédicats de typage sémantique. Wright et Felleisen [100] ont montré que ce système est sûr pour ML plus références, pour ML plus exceptions, et pour ML plus `callcc`, en utilisant dans les trois cas une sémantique par règles de réduction et un résultat de préservation du typage au cours de la réduction.

Résultats. Ce système est une amélioration considérable par-rapport à celui de Caml : on peut enfin avoir des fonctions génériques qui créent des structures modifiables ; ceci permet la fourniture en bibliothèque de nombreuses structures de données (voir [7] pour un exemple de telle bibliothèque).

Cependant, ce système a deux points faibles majeurs. Premièrement, le critère de non-expansivité est très naïf, et ne reconnaît pas comme non-expansives bien des expressions qui, sémantiquement, le sont ; ces expressions restent monomorphes, alors qu'elles devraient avoir un type polymorphe. Le phénomène se produit surtout lorsqu'on mélange fonctions d'ordre supérieur et fonctions qui créent des références. Dans l'exemple

```
let make_ref2 = (λx.x)(make_ref) in ...
```

le type de `make_ref2` reste monomorphe dans le corps du `let` : la variable α n'est pas généralisable dans son type, $\alpha^* \to \alpha^*$ `ref`, puisque l'expression $(\lambda$`x.x`$)$(`make_ref`) est expansive suivant le critère de Tofte. Clairement, `make_ref2` devrait avoir le même type faiblement polymorphe que `make_ref`, puisqu'il a la même valeur. De même, l'application partielle `map make_ref` a un type, α^* `list` $\to$ α^* `ref list` dans lequel α^* ne peut pas être généralisée, puisque l'expression `map make_ref` est expansive d'après le critère de Tofte. Néanmoins, la forme eta-expansée λ`l. map make_ref l` peut avoir un type faiblement polymorphe, puisque cette expression est non-expansive.

Deuxième point faible, le fait que certaines références ont une portée limitée n'est pas pris en compte. Si un objet a rentre en contact avec un objet b dont le type contient des variables faibles, alors les variables de types correspondantes dans a sont automatiquement affaiblies ; les variables faibles se propagent ainsi à travers les types, même si la référence dans le type de laquelle elles apparaissent a depuis longtemps disparu. Par exemple, `imp_map` se voit attribuer le type faiblement polymorphe

$$\forall \alpha^*, \beta^*. (\alpha^* \to \beta^*) \to \alpha^* \texttt{ list} \to \beta^* \texttt{ list},$$

alors que `appl_map`, qui fait exactement la même chose, reçoit le type complètement polymorphe

$$\forall\alpha,\beta.\,(\alpha\to\beta)\to\alpha\ \texttt{list}\to\beta\ \texttt{list}.$$

La différence est sensible lorsqu'on applique ces deux fonctions à des arguments polymorphes, comme l'identité et la liste vide par exemple : `appl_map id` [] a pour type β `list` avec β généralisable, alors que `imp_map id` [] a pour type β^* `list` avec β^* non généralisable, car l'expression est expansive. De même, toute fonction générique construite au-dessus de `imp_map` aura un type moins général que si `appl_map` avait été employée à la place — alors que, je le rappelle, ces deux fonctions ont la même sémantique. La raison de cette discrépance est que `imp_map` crée de manière interne deux références de types α^* `list ref` et β^* `list ref`, ce qui force les variables α^* et β^* à être faibles dans le type de `imp_map`. Que ces deux références sont locales à chaque appel de la fonction n'est pas pris en compte : le résultat de `imp_map id` [] a pour type β^* `list`, dans lequel β est toujours marquée faible ; pourtant, la référence de type β^* `list` créée par la fonction `imp_map` est devenue inaccessible, et donc il n'y a plus lieu de restreindre la généralisation de β^*.

5.2.1.3 Standard ML of New Jersey

Dans le but de détecter plus précisément le moment où les références sont créées dans les fonctions curryfiées, MacQueen a proposé une extension du système de Tofte, où les variables, au lieu d'être séparées en variables faibles et variables non faibles, sont associées à un entier, leur "degré de faiblesse", ou plus exactement leur force. Le degré d'une variable mesure le nombre d'applications de fonctions qui doivent être effectuées avant qu'on n'alloue une référence dont le type mentionne cette variable. Les variables qui n'interviennent pas dans le typage d'un `ref` ont le degré $+\infty$. Les variables qui interviennent dans le typage d'un `ref` ont pour degré le nombre d'abstractions séparant l'introduction de la variable de l'occurrence du `ref`. Les variables de degré 0 ne sont pas généralisables ; les variables de degré $n > 0$ sont généralisables, mais chaque application de fonction décrémente le degré des variables dans le type du résultat. Par exemple, la fonction :

$$\texttt{let f} = \lambda\texttt{x}.\,\lambda\texttt{y}.\,\texttt{ref(x,y)}$$

a, dans le système de MacQueen, le type $\forall\alpha^2,\beta^2.\,\alpha^2\to\beta^2\to(\alpha^2\times\beta^2)$ `ref`. L'application partielle `f(1)` a pour type $\beta^1\to(\texttt{int}\times\beta^1)$ `ref`, dans lequel β est encore généralisable, puisque de degré 1. En conséquence, l'exemple suivant est bien typé, alors qu'il est rejeté par le système de Tofte :

```
let f = λx. λy. ref(x,y) in
let g = f(1) in
    ... g(true) ... g(2) ...
```

Historique. Ce système est utilisé dans l'implémentation Standard ML of New Jersey. Il est brièvement décrit dans le manuel de référence de cette implémentation [3]. Les règles de typage n'ont jamais été publiées ; encore moins des preuves de sûreté. Le système semble avoir changé légèrement à l'occasion d'une des dernières distributions.

Résultats. Ce système se révèle à l'usage guère plus puissant que celui de Tofte. Parmi mes exemples, il n'y en a qu'un (l'application partielle de `imp_map` à l'identité) qui est reconnu correct par le système de MacQueen, mais pas par le système de Tofte. D'autres exemples d'application partielle (`appl_map make_ref`) sont toujours rejetés à tort comme mal typés, indiquant que la pleine

fonctionnalité n'est toujours pas traitée de manière satisfaisante. D'autre part, on ne tient toujours pas compte de la portée limitée de certaines références (cas de `imp_map`).

5.2.2 Les systèmes d'effets

Les trois systèmes que je vais maintenant décrire reposent, pour le contrôle des références polymorphes, sur l'ajout d'un SYSTÈME D'EFFETS au système de types. De même que le type d'une expression décrit de manière simplifiée la valeur de l'expression, l'effet d'une expression décrit de manière simplifiée les effets de bords qu'elle effectue pendant son évaluation. Les jugements de typage prennent la forme

$$E \vdash a : \tau, F$$

où F est pris dans une algèbre d'effets encore non spécifiée. Calcul des types et calcul des effets interfèrent en deux points. Premièrement, le type d'une fonction est annoté par son effet latent, c'est-à-dire par une description des effets de bords qu'elle effectue lorsqu'elle est appliquée. Les types fonctionnels sont donc de la forme $\tau_1 \multimap\langle F\rangle\rightarrow \tau_2$, où F est l'effet latent. (Malgré les similitudes de notation, il y a une différence majeure entre le typage des fermetures et les systèmes d'effets : le type de fermeture est une information statique : il décrit ce que la valeur fonctionnelle contient déjà ; alors que l'effet latent est une information dynamique : il décrit ce que la fonction va faire une fois appliquée.) La règle de typage des fonctions illustre cette interférence :

$$\frac{E, x : \tau_1 \vdash a : \tau_2, F}{E \vdash \lambda x.\, a : (\tau_1 \multimap\langle F\rangle\rightarrow \tau_2), \emptyset}$$

(On a noté Ø l'effet nul : celui qu'on donne aux expressions sans effets de bord.) Deuxième interférence : l'effet d'une expression est pris en compte au moment où l'on généralise son type, de manière à éviter la création de références polymorphes. Comme on va maintenant le voir, des critères de généralisation plus ou moins fins peuvent être appliqués suivant l'algèbre d'effets employée.

Contexte. La description que je donne ici des systèmes d'effets est incomplète, et ne considère que l'aspect "contrôle des références polymorphes" dans un cadre d'inférence de types. Les systèmes d'effets ont été introduits par Lucassen et Gifford dans leur langage FX [29, 53] principalement pour aider à la parallélisation automatique des programmes impératifs : deux expressions sans effets de bords, ou dont les effets de bords s'effectuent sur des ensembles disjoints de références, peuvent être évaluées en parallèle sans risque de changer le comportement du programme. De plus, le langage FX est à l'origine un langage avec types et effets explicites dans le programme source ; les problèmes d'inférence des types et des effets n'ont été abordés que beaucoup plus tard [40, 89], et ont imposé des restrictions supplémentaires sur l'algèbre de types et d'effets. Dans la discussion qui suit, je ne considère que les systèmes d'effets pour lesquels existe un algorithme d'inférence. □

5.2.2.1 Effets simples

Dans les cas les plus simples, un effet est un ensemble de constantes et de variables d'effets :

$$\begin{array}{rcl} F & ::= & \{f_1, \ldots, f_n\} \cup \varsigma_1 \cup \ldots \cup \varsigma_k \\ f & ::= & \texttt{alloc} \mid \texttt{read} \mid \texttt{write} \end{array}$$

Ici, la constante `alloc` signifie que l'expression crée des références, `read`, qu'elle en lit, `write`, qu'elle en modifie. On a noté ς les variables d'effets, qui introduisent une notion de polymorphisme sur les effets similaire au polymorphisme sur les types de ML. Le calcul des effets est, avec cette algèbre d'effets, une forme un peu plus fine de l'analyse de pureté ("cette expression a-t'elle des effets de bords ?"). La généralisation du type d'une expression n'est alors autorisée que si l'expression est pure, c'est-à-dire si elle a l'effet $\emptyset$:

$$\frac{E \vdash a_1 : \tau_1, \emptyset \qquad \{\alpha_1 \dots \alpha_n\} = \mathcal{L}(\tau_1) \setminus \mathcal{L}(E) \qquad E + x \mapsto \forall \alpha_1 \dots \alpha_n . \tau_1 \vdash a_2 : \tau_2, F}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2, F}$$

$$\frac{E \vdash a_1 : \tau_1, F_1 \qquad F_1 \neq \emptyset \qquad E + x \mapsto \tau_1 \vdash a_2 : \tau_2, F_2}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2, (F_1 \cup F_2)}$$

Le typage simple des références est alors sûr : avec la restriction ci-dessus sur la généralisation, une valeur généralisée ne peut contenir de références nouvellement allouées, et donc on ne généralise jamais des variables libres dans le type d'une référence.

Historique. Le système ci-dessus est sensiblement celui que décrivent Gifford et Lucassen dans [29]. Jouvelot et Gifford [40] traitent de l'inférence de types dans un système similaire. Ils affirment que du *backtracking* est nécessaire pour faire l'inférence en présence des deux règles `let` ci-dessus, et préfèrent se rabattre sur une version encore plus simple, où ne sont généralisés que les types d'expressions non-expansives, au sens de Tofte. Clairement, "non-expansif" implique "d'effet $\emptyset$", mais pas l'inverse. Aussi, cette restriction fait-elle perdre tout intérêt au système d'effets par-rapport au système de SML. Je n'applique pas cette restriction dans mes comparaisons.

Résultats. Ce système se révèle supérieur aux systèmes à base de variables faibles sur les exemples avec fonctions d'ordre supérieur. Par exemple, l'application `id make_ref` est correctement reconnue comme pure, et donc son type est généralisable. (En effet, la fonction `id` n'a pas d'effet latent, et son argument est une expression pure.) L'annotation des fonctions par leur effet latent permet de détecter plus précisément le moment où les références sont créées. Par exemple, le type de `appl_map` est :

$$\texttt{appl\_map} : \forall \alpha, \beta, \varsigma . \ (\alpha \multimap_{\langle \varsigma \rangle} \beta) \multimap_{\langle \emptyset \rangle} \alpha \texttt{ list} \multimap_{\langle \varsigma \rangle} \beta \texttt{ list}.$$

(Les variables d'effets se généralisent exactement comme les variables de types.) L'application partielle `appl_map make_ref` a donc l'effet $\emptyset$, et son type peut être généralisé.

D'un autre côté, les fonctions génériques qui allouent des références locales ne sont pas mieux traitées qu'en SML. L'allocation, bien que purement locale, laisse une trace dans leur effet latent. Elles n'ont donc pas le même type que leurs équivalents purement applicatifs. Par exemple :

$$\begin{aligned} \texttt{appl\_map} &: \forall \alpha, \beta, \varsigma . \ (\alpha \multimap_{\langle \varsigma \rangle} \beta) \multimap_{\langle \emptyset \rangle} \alpha \texttt{ list} \multimap_{\langle \varsigma \rangle} \beta \texttt{ list} \\ \texttt{imp\_map} &: \forall \alpha, \beta, \varsigma . \ (\alpha \multimap_{\langle \varsigma \rangle} \beta) \multimap_{\langle \emptyset \rangle} \alpha \texttt{ list} \multimap_{\langle \{\texttt{alloc}\} \cup \varsigma \rangle} \beta \texttt{ list} \end{aligned}$$

Et donc, `imp_map id []` reçoit l'effet {`alloc`}, empêchant de généraliser son type.

Bien que cela n'apparaisse pas dans la figure 5.1, ce premier système d'effets est parfois moins puissant que les systèmes à variables faibles. Par exemple,

$$\texttt{let id} = (\texttt{ref } [\,]); (\lambda \texttt{x}.\,\texttt{x}) \texttt{ in id(id)}$$

est accepté en SML, mais rejeté dans ce système d'effets : l'expression $(\texttt{ref } [\,]); (\lambda \texttt{x}.\,\texttt{x})$ a l'effet $\{\texttt{alloc}\}$, et son type ne peut donc pas être généralisé.

5.2.2.2 Effets d'allocation typés

Pour corriger cette dernière faiblesse, on peut enrichir l'algèbre d'effets de manière à enregistrer non seulement le fait qu'une expression crée une référence, mais aussi le type de la référence créée :

$$\begin{array}{rcl} F & ::= & \{f_1 \dots f_n\} \cup \varsigma_1 \cup \dots \cup \varsigma_k \\ f & ::= & \texttt{alloc}(\tau) \mid \texttt{read} \mid \texttt{write} \end{array}$$

La règle de généralisation devient alors : on ne peut pas généraliser les variables qui sont libres dans un des effets d'allocation de l'expression.

$$\frac{E \vdash a_1 : \tau_1, F_1 \qquad \{\alpha_1 \dots \alpha_n\} = \mathcal{L}(\tau_1) \setminus \mathcal{L}(F_1) \setminus \mathcal{L}(E) \qquad E + x \mapsto \forall \alpha_1 \dots \alpha_n . \tau_1 \vdash a_2 : \tau_2, F_2}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2, (F_1 \cup F_2)}$$

Historique. Un article récent de Wright [99] étudie un système similaire. Wright ne garde pas trace des effets `read` et `write` (qui ne servent à rien pour le contrôle des références), et remplace $\texttt{alloc}(\tau)$ par l'ensemble des variables libres dans τ. En conséquence, un effet est, dans son système, un ensemble de variables d'effets et de variables de types : les variables libres dans les types des références créées. Son système donne les mêmes résultats que celui qu'on vient de décrire.

Résultats. Sur mes tests, ce système se comporte exactement comme le premier système d'effets : bon traitement de la pleine fonctionnalité, mauvais traitement des références à portée locale. Il accepte en plus quelques exemples anecdotiques, comme

$$\texttt{let id} = (\texttt{ref } [\,]);\ \ (\lambda \texttt{x}.\,\texttt{x}) \texttt{ in id(id)}$$

Quand on type les effets d'allocation, l'expression $(\texttt{ref } [\,]);\ \ (\lambda \texttt{x}.\,\texttt{x})$ a pour type $\alpha \multimap\!\langle\emptyset\rangle\!\rightarrow \alpha$ et pour effet $\{\texttt{alloc}(\beta \texttt{ list ref})\}$. La variable α est donc généralisable, puisque non libre dans l'effet.

5.2.2.3 Le système de Damas

Un des tout premiers systèmes de types pour les références en ML, celui proposé par Damas en 1985, peut être vu comme un système d'effets d'allocation simplifié. Dans le système de Damas, les schémas de types pour les fonctions sont annotés par un ensemble de variables de types, les variables potentiellement libres dans les types des références allouées par la fonction. À la différence du système de Wright, les types fonctionnels situés à l'intérieur d'autres types ne sont pas annotés. Plus précisément, l'algèbre de types de Damas est de la forme :

Types	τ	$::=$	α	variable de type
		$\mid$	$\tau_1 \rightarrow \tau_2$	type fonctionnel simple
		$\mid$	$\dots$	
Schémas	σ	$::=$	$\forall \alpha_1 \dots \alpha_n . \tau$	schéma classique
		$\mid$	$\forall \alpha_1 \dots \alpha_n . \tau_1 \multimap\!\langle\beta_1, \dots \beta_k\rangle\!\rightarrow \tau_2$	schéma de type fonctionnel, annoté par son effet d'allocation

La motivation de cette restriction semble être que l'absence d'effets à l'intérieur des types simplifie considérablement l'inférence de types.

Historique. Ce système est étudié dans la thèse de Damas [21]. Damas donne une preuve de sûreté de ce système vis-à-vis d'une sémantique dénotationnelle, et propose un algorithme d'inférence de types. L'ensemble est d'une lecture difficile. En particulier, la preuve de sûreté nécessite des constructions de domaines et d'idéaux fort compliquées. Tofte [91] affirme que cette preuve est fausse. Le système de Damas n'a, à ma connaissance, jamais été utilisé dans un compilateur ML.

Résultats. Le système de Damas donne des résultats très proches de ceux de SML. En raison de l'omission des effets latents dans les types fonctionnels "internes", les exemples avec fonctions d'ordre supérieur ne sont pas bien traités (`appl_map make_ref`). La portée des références n'est pas prise en compte (`imp_map`).

5.2.2.4 Effets d'allocation typés plus régions

Pour prendre en compte le fait que des références peuvent disparaître (ou, plus exactement, devenir inaccessibles) parce qu'on est sorti de la portée des identificateurs auxquels elles étaient liées, on introduit la notion de RÉGION dans les types et dans les effets. Une région représente un ensemble de références. Les types références indiquent non seulement le type de l'objet référencé, mais aussi la région dans laquelle se trouve la référence ; de même, les effets gardent maintenant trace de la région concernée.

τ	::=	τ `ref`$_\rho$	type d'une référence dans la région ρ
	|	$\tau_1 \multimap\langle F\rangle\rightarrow \tau_2$	type de fonction
	|	$\ldots$	
F	::=	$\{f_1, \ldots, f_n\} \cup \varsigma_1 \cup \ldots \cup \varsigma_k$	effet
f	::=	`alloc`$_\rho(\tau)$	allocation d'une τ `ref` dans la région ρ
	|	`read`$_\rho$	lecture d'une des références de la région ρ
	|	`write`$_\rho$	modification d'une des références de la région ρ

Du point de vue du typage, les régions sont traitées comme des variables de types. En particulier, l'identification de deux types de références τ `ref`$_\rho$ et τ `ref`$_{\rho'}$ entraîne l'identification de ρ et ρ', c'est-à-dire la fusion des deux régions. Aussi, si deux objets références ont pour types τ `ref`$_\rho$ et τ' `ref`$_{\rho'}$ avec $\rho \neq \rho'$, cela signifie que les deux objets ne peuvent pas pointer vers la même adresse mémoire. Cette information est précieuse pour la compilation efficace et la parallélisation automatique. Elle permet aussi de savoir quand une référence devient inaccessible. Si, à un instant donné de l'évaluation, l'environnement courant ne contient plus aucune référence appartenant à la région ρ, cela veut dire que toutes les références allouées dans la région ρ sont inaccessibles à ce point. Donc, on peut ignorer les effets qui portent sur la région ρ : ces effets n'ont aucune influence sur les calculs qui suivent, puisqu'ils portent sur des références devenues inaccessibles. Ceci se traduit par la règle suivante, de simplification des effets :

$$\frac{E \vdash a : \tau, F}{E \vdash a : \tau,\ \texttt{Observe}(F, E, \tau)}$$

L'opérateur `Observe` renvoie le sous-ensemble de F constitué des effets qui portent sur des régions libres dans E ou dans τ. Ces effets-là peuvent être "observés" depuis l'extérieur ; les autres peuvent être effacés, puisqu'ils ne sont certainement pas observables.

Historique. La notion de région et la règle de "masquage d'effets" ci-dessus sont décrits dans [53], pour un langage explicitement typé, et où les régions sont explicitement déclarées et associées à chaque référence créée dans le programme source. Récemment, Talpin et Jouvelot [89] ont montré comment inférer régions et effets pour un langage source proche de ML.

Résultats. Grâce au masquage d'effets, des fonctions génériques peuvent avoir le même type, qu'elles utilisent ou non des références de manière interne. Par exemple, le typage principal de `imp_map` débouche sur un type de la forme

$$\texttt{imp\_map} : (\alpha \multimap\langle\varsigma\rangle\rightarrow \beta) \multimap\langle\emptyset\rangle\rightarrow \alpha\ \texttt{list} \multimap\langle\varsigma \cup \{\texttt{alloc}_\rho(\alpha), \texttt{alloc}_{\rho'}(\beta)\}\rangle\rightarrow \beta\ \texttt{list}$$

où, si le typage est principal, ρ et ρ' sont deux "nouvelles" régions, qui n'apparaissent nulle part ailleurs ; ceci est dû au fait que les deux références créées par `imp_map` ne sont jamais passées en argument à une autre fonction. Dès lors, la règle de masquage permet d'effacer les deux effets d'allocation, obtenant

$$\texttt{imp\_map} : (\alpha \multimap\langle\varsigma\rangle\rightarrow \beta) \multimap\langle\emptyset\rangle\rightarrow \alpha\ \texttt{list} \multimap\langle\varsigma\rangle\rightarrow \beta\ \texttt{list},$$

qui n'est autre que le type de `appl_map`.

Les propriétés de ce système de types sont proches de celles du système du chapitre 3. Il n'est donc pas surprenant qu'apparaissent les problèmes mentionnés partie 4.1 : l'exemple `eta_ref` illustre le besoin d'effets infinis (récursifs) ; l'exemple `capt_id_ref` montre un phénomène de capture de variables via les effets. Détaillons ces deux exemples.

```
let eta_ref = λf.
    let r = ref(f) in
        either f (λx.!(either r (ref(f)))(x))
```

Au moment de typer le `either` le plus externe, on a

$$\begin{aligned} \texttt{f} &: \alpha \multimap\langle\varsigma\rangle\rightarrow \alpha \\ \texttt{r} &: (\alpha \multimap\langle\varsigma\rangle\rightarrow \alpha)\ \texttt{ref}_\rho \end{aligned}$$

Le deuxième argument de `either`, (λx.!(`either r (ref(f)))(x)`), a pour type :

$$\alpha \multimap\langle\varsigma \cup \{\texttt{alloc}_\rho(\alpha \multimap\langle\varsigma\rangle\rightarrow \alpha)\}\rangle\rightarrow \alpha.$$

En effet, cette fonction alloue une référence vers `f`, et cette référence appartient forcément à la même région ρ que `r`, suite au typage du `either` interne. Comme ρ est accessible depuis l'environnement, l'effet d'allocation n'est pas masquable. Mais il n'y a pas d'effet (fini) qui soit une instance commune de ς et de $\varsigma \cup \{\texttt{alloc}_\rho(\alpha \multimap\langle\varsigma\rangle\rightarrow \alpha)\}$.

La deuxième source d'échec provient d'un phénomène de capture de certaines variables via les effets, les rendant non généralisables, comme expliqué partie 4.1.2.

```
let capt_id_ref = λf.
    let id = λy.
        let r = ref(y) in
            either f (λz. either r (ref(y)); z);
        y
    in id(id)
```

Avant typage du `either` externe, on a les types suivants :

$$\begin{array}{rcl} \mathtt{f} & : & \alpha \multimap\!\langle \varsigma \rangle\!\rightarrow \alpha \\ \mathtt{y} & : & \beta \\ \mathtt{r} & : & \beta\ \mathtt{ref}_\rho \\ (\lambda \mathtt{z}.\, \mathtt{either\ r\ (ref(y)); z}) & : & \gamma \multimap\!\langle \mathtt{alloc}_\rho(\beta) \rangle\!\rightarrow \gamma \end{array}$$

La fonction $\lambda \mathtt{z} \ldots$ alloue en effet une référence vers `y`, et dans la même région ρ que `r`, suite à l'intervention du `either` interne. L'effet d'allocation n'est pas masquable, puisque ρ est accessible à partir de `r`. L'identification des types de `f` et de $\lambda \mathtt{z} \ldots$ conduit aux types suivants :

$$\begin{array}{rcl} \mathtt{f} & : & \alpha \multimap\!\langle \varsigma \cup \{\mathtt{alloc}_\rho(\beta)\} \rangle\!\rightarrow \alpha \\ \mathtt{y} & : & \beta \\ \mathtt{r} & : & \beta\ \mathtt{ref}_\rho \end{array}$$

L'expression qui est liée à `id` a donc le type $\beta \multimap\!\langle F \rangle\!\rightarrow \beta$, pour un effet F qu'on ne détaille pas, dans l'environnement de typage $\mathtt{f} : \alpha \multimap\!\langle \varsigma \cup \{\mathtt{alloc}_\rho(\beta)\} \rangle\!\rightarrow \alpha$. La variable β est libre dans cet environnement, et donc non généralisable. Aussi `id` reste monomorphe, et l'auto-application `id(id)` échoue.

Remarque. J'ai cru un instant que le masquage d'effets suffisait à éliminer les problèmes de capture de variables via les effets latents des fonctions. Il n'en est rien, comme on vient de le voir. La règle de masquage d'effets est une règle de "garbage collection" : on peut ignorer tout ce qui se passe dans une région si cette région n'est mentionnée nulle part dans le contexte de typage courant (l'environnement plus le type). Elle ne suffit pas à rendre compte des propriétés d'abstraction des fonctions : une fonction peut faire référence à une certaine région de manière interne, sans pour autant rendre cette région accessible à l'extérieur. Le phénomène de capture de variables provient de la non prise en compte de cette propriété d'abstraction. En revanche, l'opération de simplification des types de fermetures suggérée dans la partie 4.1.2 tient compte de cette propriété d'abstraction, et est donc essentiellement différente d'une opération de "garbage collection" comme le masquage d'effets. □

5.2.3 Les systèmes de cette Thèse

Je commente ici les résultats obtenus avec l'approche que je propose : variables dangereuses plus typage des fermetures.

Résultats. Cette approche, au contraire des précédentes, ne cherche pas à capturer des informations sur l'aspect dynamique de l'évaluation ("cette fonction fait ci et ça") ; elle se contente,

comme le typage usuel, d'informations purement statiques ("cette valeur contient ci et ça"). Elle est, de ce fait, beaucoup plus proche d'un typage classique.

Il n'est donc pas surprenant que la pleine fonctionnalité ne pose aucun problème : `id make_ref` a évidemment le même type que `make_ref` ; quant à `appl_map make_ref`, le typage nous dit que c'est une valeur fonctionnelle dont la fermeture ne contient pas de références, elle reçoit donc un type complètement polymorphe.

De même, le fait que certaines références ont une portée locale se voit très bien sur les types : si une fonction crée des références, mais que ces références n'apparaissent ni dans la valeur de retour, ni dans d'autres références accessibles — ce que le typage nous dit facilement —, il est clair que ces références sont purement locales. Ainsi, `imp_map` a exactement le même type que `appl_map`, et peut donc être employé en lieu et place de `appl_map` dans tout programme.

Les exemples techniques (`eta`, `eta_ref`, `capt_id`, `capt_id_ref`) font apparaître quelques lacunes du système décrit au chapitre 3. Le système du chapitre 4, tout en conservant le bon comportement du système du chapitre 3 sur les exemples pratiques, fait disparaître la plupart de ces lacunes : les types de fermetures récursifs ne font pas échouer le typage (`eta` et `eta_ref`) ; et le phénomène de capture de variables via les types de fermetures est évité dans l'exemple `capt_id` ; c'était le but recherché. Comme le montre l'exemple `capt_id_ref`, ce phénomène de capture se manifeste encore pour certains programmes non purement fonctionnels.

```
let capt_id_ref = λf.
    let id = λy.
        let r = ref(y) in
            either f (λz. either r (ref(y)); z);
        y
    in id(id)
```

Le typage principal de $\lambda\mathtt{y}\ldots$ sous l'hypothèse $\mathtt{f} : t_n \multimap\!\langle u_n\rangle\!\rightarrowtail t_n$ débouche sur le type $t'_n \multimap\!\langle u'_n\rangle\!\rightarrowtail t'_n$ sous les contraintes

$$t'_n \ \mathtt{ref} \triangleleft u_n,\ (t_n \multimap\!\langle u_n\rangle\!\rightarrowtail t_n) \triangleleft u'_n$$

La variable t'_n est dangereuse dans l'environnement de typage, et donc non généralisable. Ce phénomène de capture semble pouvoir être contourné comme au chapitre 4 : de même qu'on peut ignorer les variables libres dans la partie type de fermeture d'un type fonctionnel si elles ne sont pas également libres dans le type de l'argument ou le type du résultat, je crois qu'il est correct d'ignorer les variables dangereuses dans la partie type de fermeture d'un type fonctionnel si elles ne sont pas également libres dans le type de l'argument ou le type du résultat. Plus formellement, on prendrait :

$$\mathcal{D}(\sigma_1 \multimap\!\langle u\rangle\!\rightarrowtail \sigma_2 \,/\, C) = \mathcal{D}(u \,/\, C) \cap (\mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2)).$$

L'intuition est que de telles variables dangereuses correspondent à des références non accessibles depuis l'extérieur de la fonction. Ainsi, dans l'exemple `capt_id_ref`, t'_n ne serait plus dangereuse dans l'environnement de typage au moment de la généralisation. Je n'ai pas essayé de prouver la sûreté de ce raffinement du système du chapitre 4.

L'exemple `fake_ref`, qui consiste à contraindre le type d'une exception pour faire croire qu'une référence polymorphe est créée, est refusé par mes systèmes, mais accepté par tous les autres.

L'importance pratique de ce fait est très exactement nulle. Il s'agit simplement de souligner la différence entre l'approche "tout dans les types, rien que les types", qui conduit à rejeter toute expression dont le type ressemble exactement au type d'une création de référence polymorphe, et les autres approches, qui ajoutent divers mécanismes au typage pour précisément mieux distinguer la création des références.

5.3 Facilité d'emploi et compatibilité avec les modules

Programmer dans un langage typé nécessite de la part du programmeur une bonne compréhension de son système de types : pour écrire les déclarations de types dans le programme source ; pour comprendre les erreurs de types signalées par le compilateur ; enfin, pour écrire les spécifications de types dans les interfaces de modules. Cet argument traditionnel milite en faveur de systèmes de types les plus simples possible. Il met une limite à la recherche de systèmes de types toujours plus expressifs, qui tend naturellement à déboucher sur des systèmes de types complexes. (La progression entre les systèmes des chapitres 1, 3 et 4 illustre bien cette tendance ; les autres systèmes présentés au début de ce chapitre s'inscrivent aussi dans cette tendance.)

L'inférence de types à la manière de ML peut sembler à première vue résoudre cette tension : si les types sont laissés implicites dans le programme source et synthétisés par le compilateur, le programmeur n'a plus à craindre la complexité du système de types. En fait, l'inférence de types rend le problème moins aigu, mais ne le fait pas disparaître. Il reste en effet un certain nombre d'occasions où le programmeur doit lire ou écrire lui-même des expressions de types.

Premièrement, lorsqu'une erreur de types est détectée, il faut déchiffrer les types inférés par le compilateur, comprendre pourquoi ils rentrent en conflit, et de quel morceau du code provient l'erreur. Le dernier point est souvent difficile, même avec un système de types simple. Un système plus complexe aggrave la situation, sans pour autant la rendre désespérée. Le compilateur peut en effet simplifier les expressions de types qu'il présente à l'utilisateur, tant que les types simplifiés font toujours apparaître le conflit de types. Par exemple, dans le cas des systèmes qui annotent les types de fonctions par des informations supplémentaires (comme ceux des chapitres 3 et 4, mais aussi comme les systèmes d'effets présentés plus haut), on peut ne pas afficher ces informations supplémentaires quand il s'agit d'une application de fonction à un objet du mauvais type — erreur de loin la plus fréquente. D'autre part, les erreurs de types vraiment subtiles, et tout particulièrement celles qui ont rapport avec les références/canaux/continuations polymorphes, sont rares, aussi bien de la part de programmeurs novices (qui ne savent de toute façon pas mélanger traits non-applicatifs et pleine fonctionnalité) que de programmeurs expérimentés (qui font rarement d'erreurs de types). On peut donc admettre que ces erreurs subtiles demandent un certain effort de réflexion de la part du programmeur.

Les situations où le programmeur doit écrire des types à la main posent des problèmes plus graves. Comme exemples de telles situations, en ML, on a les déclarations de structures de données (types concrets), où le programmeur doit donner les types des arguments des constructeurs, et la spécification des interfaces (ou "signatures") de modules, où le programmeur doit spécifier les types des identificateurs globaux exportés par le module. Dans le cas où seuls les types de fonctions sont rendus plus complexes, comme dans tous les systèmes considérés plus haut, le problème des déclarations de types concrets est mineur : rares sont en pratique les types concrets contenant des

valeurs fonctionnelles. Le problème des interfaces de modules est en revanche crucial : la plupart des valeurs exportées par un module sont des fonctions.

Le problème de la facilité d'utilisation des systèmes de types considérés ici se réduit donc au problème suivant : à quel point est-il difficile de spécifier complètement, dans une interface de module, le type d'une fonction dont généralement on ne connaît pas l'implémentation ? (Dans le cas où on écrit l'implémentation avant l'interface, on peut toujours lancer l'inféreur de types sur l'implémentation, puis recopier dans l'interface les types inférés par "couper-coller". Une telle pratique est contraire aux principes de la programmation modulaire [70, 12], qui veulent que la spécification précède l'implémentation. Elle ne s'applique de toute façon pas à l'écriture de l'interface pour les paramètres de modules paramétrés, ou "foncteurs".)

Sur les systèmes présentés dans la partie précédente, il semble que cette difficulté est à peu près proportionnelle à l'expressivité du système : plus le système est expressif, plus ses expressions de types reflètent la manière dont une fonction est implémentée, et plus il est difficile de spécifier le type d'une fonction sans l'avoir implémentée.

La progression est frappante dans le cas des systèmes à variables faibles. Le système de CAML, le plus restrictif de tous, utilise la même algèbre de types que le noyau ML, et donc se révèle tout aussi adapté à l'écriture d'interfaces. Le système de Standard ML, plus expressif, distingue variables applicatives et variables impératives dans les types ; il faut donc, pour spécifier le type d'une fonction polymorphe, savoir si elle va être implémentée dans le style applicatif (auquel cas toutes les variables de son type peuvent être applicatives) ou bien dans le style impératif (auquel cas certaines variables doivent être impératives). L'introduction de niveaux de faiblesse, comme en SML-NJ, renforce encore ce problème : ces niveaux dépendent étroitement de la manière dont les fonctions curryfiées entrelacent calculs et passage d'arguments ; par conséquent, la spécification de type d'une fonction polymorphe curryfiée impose non seulement le style (purement applicatif ou non) dans lequel elle est implémentée, mais encore l'ordre dans lequel calculs et passages d'arguments sont effectués. Clairement, il est difficile d'imaginer a priori ces propriétés tant que l'implémentation n'est pas écrite.

Les systèmes d'effets semblent suivre une progression similaire.[1] Avec une algèbre d'effets simple, la spécification du type d'une fonction indique non seulement son style d'implémentation, mais aussi, dans le cas d'une fonction d'ordre supérieur, à quel moment les fonctions prises en argument sont appliquées (ceci se reflète dans les positions des variables d'effets représentant les effets latents de ces paramètres fonctionnels). Avec une algèbre d'effets enrichie de régions, la spécification de type doit en plus exprimer les propriétés de partage de valeurs de l'implémentation. En effet, les parties "régions" des types traduisent très finement comment une implémentation partage des références entre arguments et résultats. Ces propriétés de partage dépendent étroitement de l'implémentation, et il semble donc très difficile de les mettre dans la spécification de types.

Mon approche — le typage des fermetures — ajoute elle aussi des obstacles majeurs à l'écriture d'interfaces de modules. De manière générale, il est difficile d'imaginer quels types de fermetures mettre au-dessus des flèches du type d'une fonction qui n'est pas encore implémentée. De plus,

[1] N'ayant pas pu acquérir d'expérience pratique avec une implémentation d'un langage à système d'effets, je ne peux donner ici que des impressions a priori. L'abondante littérature sur les systèmes d'effets [29, 53, 40, 99, 89] ne dit rien sur les problèmes propres à la programmation modulaire.

deux implémentations d'une fonction curryfiée peuvent choisir d'entrelacer différemment calculs et passage d'arguments, ce qui débouche sur deux types différents pour deux implémentations qu'on voudrait sans doute considérer comme équivalentes. Remarquons que ce problème ne se manifeste pas dans le cas des fonctions monomorphes, cas de loin le plus fréquent. En effet, on peut omettre, dans les spécifications de types de fermetures, tous les types clos ; et donc, une spécification de fonction monomorphe ne contient que des types de fermeture triviaux. Ce problème ne se manifeste pas non plus pour des fonctions éventuellement polymorphes, mais non curryfiées : des fonctions de la forme `let f (x,y,z)` $= a$, où a ne renvoie pas de valeur fonctionnelle. Ces fonctions, elles aussi, se contentent d'un type de fermeture trivial. Mais force m'est de reconnaître que des types de fermeture non triviaux (et même souvent compliqués) sont nécessaires pour les fonctions polymorphes curryfiées, et tout particulièrement pour celles qui prennent des paramètres fonctionnels. Un examen rapide de quelques bibliothèques ML [7, 98, 57] montre que de nombreuses fonctions utiles tombent dans cette catégorie. Beaucoup de ces fonctions pourraient être décurryfiées sans changement de sémantique ; mais la forme curryfiée semble avoir la préférence des programmeurs ML. (Je ne le leur reproche pas : j'ai moi-même argumenté vigoureusement par ailleurs [49] en faveur de la curryfication, et conçu le modèle d'exécution de Caml Light spécialement pour que les fonctions curryfiées s'exécutent plus rapidement que leurs formes décurryfiées.) Pour ce type de fonctions, le typage des fermetures entre nettement en conflit avec les impératifs de la programmation modulaire.

Chapitre 6

Polymorphisme par nom

Dans ce chapitre, on montre que les difficultés rencontrées dans le typage polymorphe des références, canaux et continuations sont essentiellement liées à la sémantique de la généralisation et de la spécialisation en ML : si on donne à ces constructions une autre sémantique, que j'appelle la sémantique du polymorphisme par nom, les règles de typage naïves pour les références, les canaux et les continuations se révèlent correctes. Ce résultat suggère qu'une variante de ML avec polymorphisme par nom pourrait être un langage mieux adapté que ML au maniement des références, canaux, et continuations. La fin du chapitre discute d'un point de vue pratique les forces et les faiblesses de cette variante.

6.1 Présentation informelle

6.1.1 Les sémantiques du polymorphisme

Le polymorphisme s'introduit par le biais de deux opérations de base : la généralisation, qui transforme un terme de type $\tau[\alpha]$, où α est une inconnue de typage, en un terme de type $\forall\alpha.\,\tau[\alpha]$; et la spécialisation, qui transforme un terme de type $\forall\alpha.\,\tau[\alpha]$ en un terme de type $\tau[\tau']$, pour n'importe quel type τ'. En ML, ces deux opérations n'apparaissent pas explicitement dans le texte du programme, et sont effectuées implicitement à certains points du programme (dans la construction `let`, pour la généralisation ; à l'accès à une variable, pour la spécialisation). Dans d'autres langages, ces deux opérations sont explicites dans les programmes : il y a des constructions syntaxiques pour généraliser et pour spécialiser. Par exemple, dans les lambda-calculs polymorphes de Girard et de Reynolds [30, 81], la généralisation est présentée comme une abstraction sur une variable de type. On la note $\Lambda\alpha.\,a$, par analogie avec la notation pour les fonctions $\lambda x.\,a$. Symétriquement, la spécialisation est présentée comme l'application d'un terme à un type. On la note $a\langle\tau\rangle$, par analogie avec l'application de fonctions. Des langages comme Poly [56] et Quest [12] suivent la même approche.

Quelle que soit la présentation syntaxique retenue (explicite ou implicite), il y a deux sémantiques possibles pour les constructions de généralisation et de spécialisation. La première possibilité est de considérer que ces deux constructions n'ont pas de contenu calculatoire. En notation lambda-calcul

polymorphe, ceci revient à dire que $\Lambda\alpha.\, a$ se comporte à l'évaluation exactement comme a, et de même $a\langle\tau\rangle$ se comporte comme a :

$$\frac{e \vdash a \Rightarrow r}{e \vdash \Lambda\alpha.\, a \Rightarrow r} \qquad \frac{e \vdash a \Rightarrow r}{e \vdash a\langle\tau\rangle \Rightarrow r}$$

En notation ML, dans le cadre de l'évaluation stricte, ceci revient à dire que l'expression `let` $x = a_1$ `in` a_2 évalue a_1 une fois pour toutes, et partage la valeur obtenue entre toutes les occurrences de x dans a_2. C'est ce qui se passe en ML, et dans le petit langage étudié dans les chapitres précédents.

L'autre sémantique possible est d'interpréter la généralisation comme une abstraction fonctionnelle, et la spécialisation comme une application de fonction. En d'autres termes, la généralisation suspend l'évaluation de l'expression généralisée, et chaque spécialisation la ré-évalue. En notation lambda-calcul polymorphe, ceci revient à traiter Λ comme une "vraie" abstraction (qui construit une suspension) et $\langle\cdot\rangle$ comme une "vraie" application (qui évalue la suspension) :

$$e \vdash \Lambda\alpha.\, a \Rightarrow \mathtt{Susp}(a, e) \qquad \frac{e \vdash a \Rightarrow \mathtt{Susp}(a', e') \qquad e' \vdash a' \Rightarrow r}{e \vdash a\langle\tau\rangle \Rightarrow r}$$

C'est ce qui se passe, par exemple, en Quest. Les "génériques" de CLU [52] ou de Ada [94], qui peuvent être vus comme une forme restreinte de polymorphisme, ont également cette sémantique.

Je parle de POLYMORPHISME PAR VALEUR pour la première interprétation, et de POLYMORPHISME PAR NOM pour la seconde interprétation, par analogie avec les deux sémantiques bien connues de l'appel de fonction : l'appel par valeur et l'appel par nom[1](comme en Algol 60).

6.1.2 Polymorphisme par nom en ML

On pourrait croire que la sémantique du polymorphisme par nom n'est possible que dans un langage avec polymorphisme explicite dans la syntaxe. Il n'en n'est rien, et je vais le montrer dans ce chapitre en étudiant une variante de ML — avec polymorphisme implicite et inférence de types, donc — où le polymorphisme est interprété suivant la sémantique "par nom". (Rouaix a utilisé un langage similaire pour étudier la surcharge dynamique [86, 85].) L'idée de départ est de séparer les deux rôles du `let` en ML classique : premier rôle, l'introduction de types polymorphes ; deuxième rôle, le partage de la valeur d'une expression entre plusieurs utilisations. On remplace donc `let` par deux constructions syntaxiques. La première est le `let` strict mais monomorphe :

$$\mathtt{let\ val}\ x = a_1\ \mathtt{in}\ a_2.$$

[1]Dans les langages modernes, l'appel par nom est souvent délaissé au profit d'une variante, l'appel par nécessité (aussi appelé "évaluation paresseuse"), où les arguments de fonctions sont évalués la première fois qu'on en a besoin, et où le résultat obtenu est réutilisé par la suite. C'est la stratégie employée dans la plupart des implémentations de langages purement fonctionnels comme Haskell, Miranda ou Lazy ML [4, 72]. Cette stratégie, plus efficace, est sémantiquement équivalente à l'appel par nom dans un langage purement applicatif. Ce n'est plus le cas dans un langage avec des traits impératifs. Dans l'étude qui va suivre du polymorphisme par nom dans un langage non purement applicatif, il est essentiel de ré-évaluer l'objet polymorphe à chaque spécialisation, sans partager la valeur obtenue entre plusieurs spécialisations.

Cette construction évalue a_1 une seule fois, et partage la valeur obtenue entre toutes les occurrences de x dans a_2. Mais elle ne généralise pas le type de a_1. En d'autres termes, l'expression `let val` ci-dessus s'évalue et se type exactement comme $(\lambda x.\, a_2)(a_1)$. (On gardera la notation `let val` dans les exemples, pour plus de lisibilité.) L'autre construction est le `let` polymorphe mais par nom :

$$\texttt{let poly } x = a_1 \texttt{ in } a_2.$$

Cette construction généralise le type de a_1 comme à l'ordinaire. Mais elle n'évalue pas a_1 une fois pour toutes ; au contraire, a_1 est ré-évalué à chaque fois qu'on accède à x dans a_2. En d'autres termes, l'expression `let poly` ci-dessus s'évalue exactement comme la substitution textuelle $[x \mapsto a_1](a_2)$.

Contexte. L'expression `let poly` pourrait aussi se typer presque exactement comme $[x \mapsto a_1](a_2)$. En effet, c'est une propriété bien connue du système de types de Milner que, dans le cas non dégénéré où x apparaît dans a_2, l'expression $\texttt{let } x = a_1 \texttt{ in } a_2$ a le type τ si et seulement si la substitution textuelle $[x \mapsto a_1](a_2)$ a le type τ [66, section 4.7.2]. Certains auteurs s'appuient sur cette propriété pour donner, du système de types de Milner, une présentation simplifiée qui n'utilise pas de schémas de types [44, 100]. Je n'ai pas utilisé cette présentation dans ce travail, car elle ne débouche pas directement sur un algorithme d'inférence de types raisonnablement efficace. □

6.1.3 Polymorphisme par nom et constructions impératives

Lorsqu'on interprète le polymorphisme suivant la sémantique par nom, le typage polymorphe de constructions impératives comme les références, les canaux et les continuations ne pose plus aucun problème : le typage naïf de ces constructions se révèle sémantiquement correct. (Par "typage naïf", j'entends les types donnés dans les parties 2.1.3, 2.2.3 et 2.3.3.) Ce qui pose problème quand on combine polymorphisme et constructions impératives, c'est la possibilité d'utiliser un même objet référence/canal/continuation avec plusieurs types différents. Mais cette situation ne peut pas se produire si on adopte la sémantique par nom pour le polymorphisme. Pour employer un tel objet avec plusieurs types différents, il faut le lier à une variable (pour pouvoir y faire référence plusieurs fois). Si la liaison se fait par un λ ou par un `let val`, la variable reste monomorphe, et est donc toujours employée avec le même type. Et si la liaison se fait par un `let poly`, chaque utilisation de la variable ré-évalue l'expression à laquelle elle est liée, re-créant ainsi un objet référence/canal/continuation différent à chaque fois ; et donc, on n'a toujours pas pu accéder au même objet avec plusieurs types différents.

Pour illustrer ceci, on va maintenant reprendre certains exemples des chapitres 2 et 3, en adoptant la sémantique par nom pour le polymorphisme.

Exemple. Commençons par le pont-aux-ânes des références polymorphes :

```
let r = ref(λx.x) in
    r := (λn.n + 1);
    if (!r)(true) then ... else ...
```

Avec la sémantique par nom, il faut remplacer le `let` initial soit par un `let val`, soit par un `let poly`. Si on met un `let val` strict, le type de `r`, $(\alpha \to \alpha)$ `ref`, n'est pas généralisé pendant le typage du corps du `let val`. Le typage de l'affectation instancie α en `int`, et l'application

(!r)(true) est rejetée à la compilation comme mal typée. Si on met un `let poly`, le programme est bien typé (avec `r` de type $\forall\alpha.\,(\alpha \to \alpha)$ `ref` dans le corps du `let poly`). Mais, à l'exécution, chacun des deux accès à `r` ré-évalue l'expression `ref`(λx. x), créant ainsi deux références différentes à la fonction identité. L'affectation modifie la première de ces références, mais l'application (!r)(true) consulte la deuxième, qui pointe toujours vers la fonction identité. L'exemple s'exécute donc sans erreurs. □

La sémantique du polymorphisme par nom élimine les utilisations pathologiques de références polymorphes. En revanche, elle ne s'oppose en rien aux utilisations raisonnables de références à l'intérieur de fonctions génériques.

Exemple. La fonction qui renverse une liste de manière itérative s'écrit comme suit :

```
let poly reverse = λl.
    let val arg = ref(l) in
    let val res = ref(nil) in
        while not is_null(!arg) do
            res := cons(head(!arg), !res);
            arg := tail(!arg)
        done;
        !res
```

Les deux `let val` assurent que tous les accès à `arg` renvoient la même référence, et de même pour `res`, ce qui est indispensable au bon déroulement de la boucle. Toutes les occurrences de `arg` et `res` dans la boucle ont le même type α `list ref` (si α `list` est le type de `l`) ; la boucle est donc bien typée. Le `let poly` externe donne à `reverse` le type polymorphe désiré : $\forall\alpha.\,\alpha$ `list` $\to \alpha$ `list`. Le fait que la fermeture λl... est recalculée à chaque accès à `reverse`, au lieu d'être partagée comme dans le cas du polymorphisme par valeur, ne change pas le comportement de `reverse`. □

Exemple. Le contre-exemple à base de canaux se comporte de la même manière que le contre-exemple à base de références :

```
let c = newchan() in (c!true) ‖ (1 + c?)
```

Si le `let` est transformé en `let val`, l'exemple est rejeté au typage. Si le `let` est transformé en `let poly`, les deux accès à `c` renvoient des canaux différents, et donc les deux processus ne peuvent pas communiquer, ils restent bloqués jusqu'à la nuit des temps. L'important est qu'aucune erreur de types ne se produit à l'exécution. (Le comportement de blocage n'est bien sûr pas ce que l'auteur du programme avait en tête ; mais après tout, ce programme est clairement erroné.) □

Exemple. Traitons pour finir l'exemple de Harper et Lillibridge pour les continuations :

```
let later =
    callcc(λk.
        (λx. x),
        (λf. throw(k, (f, λx. ()))))
in
    print_string(first(later)("Hello !"));
    second(later)(λx. x + 1)
```

Le programme n'est pas statiquement bien typé si on remplace `let` par `let val`. Si on remplace `let` par `let poly`, le `callcc` est évalué non plus une seule fois juste avant d'évaluer le corps du `let`, mais deux fois, à l'occasion de chacun des deux accès à `later`. La première fois, la continuation capturée est λ`later. print_string`..., et elle n'est pas activée. La deuxième fois, la continuation capturée est λ`later. second(later)(`λ`x. x + 1)`. L'évaluation de `second(later)(`λ`x. x + 1)` relance cette continuation sur la valeur (λ`x. x + 1, `λ`x. ()`). On va donc évaluer à nouveau `second(later)(`λ`x. x + 1)`, cette fois dans un environnement où `later` est lié à (λ`x. x + 1, `λ`x. ()`) — en fait, à une suspension qui s'évalue en cette paire de fonctions. L'évaluation finit par renvoyer `()`, sans avoir provoqué d'erreur de types. □

Contexte. On se convainc aisément que le typage polymorphe des constructions impératives est sûr si on emploie la sémantique du polymorphisme par nom ; on le démontre sans difficultés, comme le montre la prochaine partie. Pourtant, ce résultat semble peu connu. Gifford et Lucassen y font allusion dans [29]. Cardelli sous-entend également ce fait dans sa description du langage Quest [12]. Dans les deux cas, la discussion porte sur des langages avec polymorphisme explicite, et tombe dans la confusion entre polymorphisme explicite et sémantique par nom du polymorphisme. Par exemple, Cardelli écrit [12, p. 24] :

> Mutability [in Quest] interacts very nicely with all the quantifiers, including polymorphism, showing that the functional-style approach suggested by type theory does not prevent the design of imperative languages.

(Bien dit.) Et il ajoute dans une note de bas de page d'une brièveté bien à lui :

> The problems encountered in ML are avoided by the use of explicit polymorphism.

Ce n'est pas exact : si les structures mutables ne posent pas de problèmes en Quest, c'est parce que le polymorphisme y est interprété avec la sémantique par nom, et non pas parce qu'il est explicite dans les programmes source. Cette confusion entre la syntaxe et la sémantique du polymorphisme est naturellement entretenue par le fait que la sémantique par valeur et la sémantique par nom du polymorphisme sont indistinguables dans le lambda-calcul typé polymorphe, du fait de la propriété de normalisation forte ; les différences entre les deux sémantiques ne se manifestent que lorsqu'on ajoute la récursion, ou des constructions non purement fonctionnelles.

□

6.2 Sémantique opérationnelle

Dans cette partie, on définit formellement la sémantique opérationnelle des trois calculs (celui avec les références, celui avec les canaux, celui avec les continuations) lorsqu'on interprète le polymorphisme par nom. Les trois sémantiques sont très proches de celles données au chapitre 2. Pour simplifier la présentation, on partage les identificateurs de variables en deux classes : les identificateurs stricts (ensemble `SVar`, notation x_s), qui sont liés par l'abstraction de fonction (f_s `where` $f_s(x_s) = a$), et les identificateurs retardés (ensemble `RVar`, notation x_r), qui sont liés par la construction `let poly`.

Pour ce qui est des objets sémantiques, la principale différence porte sur les environnements d'évaluation : ils associent aux variables strictes une valeur, et aux variables retardées une suspension, c'est-à-dire un couple (a, e) d'une expression non évaluée a et de son environnement d'évaluation e. Pour ce qui est des règles d'évaluation, les règles sont identiques à celles du chapitre 2, à l'exception de celles pour le `let` et pour l'accès à une variable.

6.2.1 Cas des références

On résume les objets sémantiques utilisés :

Résultat :	r	::=	v/s	résultat normal
		|	`err`	résultat d'erreur
Valeurs :	v	::=	cst	valeur de base
		|	(v_1, v_2)	paire de valeurs
		|	(f_s, x_s, a, e)	valeur fonctionnelle
		|	ℓ	adresse mémoire
Environnements :	e	::=	$[x_s \mapsto v, \ldots, x_r \mapsto (a, e), \ldots]$	
Etats mémoire :	s	::=	$[\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n]$	

Le prédicat d'évaluation est défini par les mêmes règles que celles de la partie 2.1.2, à l'exception de la règle pour les variables et de la règle pour le `let`. Les deux règles pour les variables sont remplacées par les quatre règles suivantes :

$$\frac{x_s \in \mathrm{Dom}(e)}{e \vdash x_s/s \Rightarrow e(x_s)/s} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x/s \Rightarrow \texttt{err}} \quad (x \text{ est } x_s \text{ ou } x_r)$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0/s \Rightarrow r}{e \vdash x_r/s \Rightarrow r}$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) \text{ n'est pas de la forme } (a, e)}{e \vdash x_r/s \Rightarrow \texttt{err}}$$

Les deux règles pour la construction `let` sont remplacées par la règle suivante :

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2/s \Rightarrow r}{e \vdash (\texttt{let poly } x_r = a_1 \texttt{ in } a_2)/s \Rightarrow r}$$

6.2.2 Cas des canaux

Résumé des objets sémantiques utilisés :

Résultats :	r	::=	v	résultat normal (une valeur)
		|	`err`	résultat d'erreur
Valeurs :	v	::=	cst	valeur de base
		|	(v_1, v_2)	paire de valeurs
		|	(f_s, x_s, a, e)	valeur fonctionnelle (fermeture)
		|	c	identificateur de canal
Environnements :	e	::=	$[x_s \mapsto v, \ldots, x_r \mapsto (a, e), \ldots]$	
Événements :	evt	::=	$c\,?\,v$	émission d'une valeur
		|	$c\,!\,v$	réception d'une valeur
Suites d'événements :	w	::=	ε	la suite vite
		|	$evt \ldots evt$	

Les règles d'évaluation sont celles de la partie 2.2.2, à l'exception des deux règles pour les accès aux variables, qui deviennent :

$$\frac{x_s \in \mathrm{Dom}(e)}{e \vdash x_s \overset{\varepsilon}{\Longrightarrow} e(x_s)} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x \overset{\varepsilon}{\Longrightarrow} \texttt{err}}$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0 \overset{u}{\Longrightarrow} r}{e \vdash x_r \overset{u}{\Longrightarrow} r}$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) \text{ n'est pas de la forme } (a, e)}{e \vdash x_r \overset{\varepsilon}{\Longrightarrow} \texttt{err}}$$

et des deux règles pour le `let`, qui deviennent :

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2 \overset{u}{\Longrightarrow} r}{e \vdash \texttt{let poly } x_r = a_1 \texttt{ in } a_2 \overset{u}{\Longrightarrow} r}$$

6.2.3 Cas des continuations

Résumé des objets sémantiques utilisés :

Résultats :	r	::=	v	résultat normal (une valeur)
		|	`err`	résultat d'erreur
Valeurs :	v	::=	cst	valeur de base
		|	(v_1, v_2)	paire de valeurs
		|	(f_s, x_s, a, e)	valeur fonctionnelle
		|	k	continuation
Environnements :	e	::=	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	
Continuations :	k	::=	`stop`	fin du programme
		|	`primc`(op, k)	après l'argument d'une primitive
		|	`app1c`(a, e, k)	après la partie fonction d'une application
		|	`app2c`(f_s, x_s, a, e, k)	après la partie argument d'une application
		|	`pair1c`(a, e, k)	après le premier argument d'une paire
		|	`pair2c`(v, k)	après le second argument d'une paire

Remarque. La continuation `letc`(x, a, e, k) n'est plus utilisée. □

Les règles d'évaluation sont celles de la partie 2.3.2, à l'exception des règles pour les accès aux variables, qui sont remplacées par :

$$\frac{x_s \in \mathrm{Dom}(e) \qquad \vdash e(x) \triangleright k \Rightarrow r}{e \vdash x_s; k \Rightarrow r} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x; k \Rightarrow \texttt{err}}$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0; k \Rightarrow r}{e \vdash x_r; k \Rightarrow r}$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) \text{ n'est pas de la forme } (a, e)}{e \vdash x_r \Rightarrow \texttt{err}}$$

et de la règle pour le `let`, qui devient :

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2; k \Rightarrow r}{e \vdash (\texttt{let poly } x_r = a_1 \texttt{ in } a_2); k \Rightarrow r}$$

6.3 Preuves de sûreté

Dans cette partie, on montre que le système de types présenté au chapitre 1 est sûr pour les trois sémantiques présentées dans la partie précédente. Les preuves sont une adaptation des preuves de sûreté de la partie 3.3 à la nouvelle sémantique du `let` et de l'accès à une variable. Les preuves utilisent les mêmes techniques, mais sont grandement simplifiées par le fait que toutes les valeurs sont maintenant monomorphes : il n'y a plus que les suspensions qui appartiennent à plusieurs types. En conséquence, il n'y a plus besoin du (délicat) lemme de généralisation sémantique.

6.3.1 Cas des références

On utilise les relations de typage sémantique suivantes :

$S \models v : \tau$	v, considérée dans un état mémoire de type S, est une valeur correcte du type τ
$S \models (a, e) : \sigma$	la suspension (a, e), considérée dans un état mémoire de type S, est une valeur correcte de toutes les instances du schéma σ
$S \models e : E$	les valeurs contenues dans l'environnement d'évaluation e, considérées dans un état mémoire de type S, appartiennent bien aux schémas correspondants dans E
$\models s : S$	l'état mémoire s est bien du type S.

Voici leurs définitions exactes :

- $S \models cst : \texttt{unit}$ si cst est `()`
- $S \models cst : \texttt{int}$ si cst est un entier
- $S \models cst : \texttt{bool}$ si cst est `true` ou `false`
- $S \models (v_1, v_2) : \tau_1 \times \tau_2$ si $S \models v_1 : \tau_1$ et $S \models v_2 : \tau_2$
- $S \models \ell : \tau$ `ref` si $\ell \in \mathrm{Dom}(S)$ et $\tau = S(\ell)$
- $S \models (f_s, x_s, a, e) : \tau_1 \to \tau_2$ s'il existe un environnement de typage E tel que

$$S \models e : E \quad \text{et} \quad E \vdash (f_s \ \texttt{where}\ f_s(x_s) = a) : \tau_1 \to \tau_2$$

- $S \models (a, e) : \forall \alpha_1 \dots \alpha_n . \tau$ si pour toute substitution φ de domaine inclus dans $\{\alpha_1 \dots \alpha_n\}$, il existe un environnement de typage E tel que

$$S \models e : E \quad \text{et} \quad E \vdash a : \varphi(\tau)$$

- $S \models e : E$ si $\mathrm{Dom}(e) = \mathrm{Dom}(E)$, et pour tout $x_s \in \mathrm{Dom}(e)$, $E(x_s)$ est un type simple τ tel que $S \models e(x_s) : \tau$, et pour tout $x_r \in \mathrm{Dom}(e)$, $E(x_r)$ est un schéma σ tel que $S \models e(x_r) : \sigma$.
- $\models s : S$ si $\mathrm{Dom}(s) = \mathrm{Dom}(S)$, et pour tout $\ell \in \mathrm{Dom}(s)$, on a $S \models s(\ell) : S(\ell)$.

Proposition 6.1 (Sûreté forte pour les références) *Soient a une expression, τ un type, E un environnement de typage, e un environnement d'évaluation, s un état mémoire, S un typage de la mémoire tels que :*

$$E \vdash a : \tau \quad \textit{et} \quad S \models e : E \quad \textit{et} \quad \models s : S.$$

S'il existe une réponse r telle que $e \vdash a/s \Rightarrow r$, alors $r \neq \texttt{err}$; au contraire, r est de la forme v/s', et il existe un typage de la mémoire S' tel que :

$$S' \textit{ prolonge } S \quad \textit{et} \quad S' \models v : \tau \quad \textit{et} \quad \models s' : S'.$$

Démonstration : la preuve procède par récurrence sur la taille de la dérivation d'évaluation. On raisonne par cas sur a, et donc sur la dernière règle utilisée dans la dérivation de typage. On donne les seuls cas qui diffèrent nettement des cas correspondants de la preuve de la proposition 3.6.

• **Cas d'une variable stricte.**

$$\frac{\tau \leq E(x_s)}{E \vdash x_s : \tau}$$

On sait par le typage que x_s appartient au domaine de E, qui est aussi le domaine de e. Donc, la seule évaluation possible est $e \vdash x_s/s \Rightarrow e(x_s)/s$. Par hypothèse sur e et E, on a $S \models e(x_s) : E(x_s)$. Comme $E(x_s)$ est un type simple, on a $\tau = E(x_s)$. D'où $S \models e(x) : \tau$. $S' = S$ convient.

• **Cas d'une variable retardée.**

$$\frac{\tau \leq E(x_r)}{E \vdash x_r : \tau}$$

Écrivons $E(x_r) = \forall \alpha_1 \ldots \alpha_n . \tau_x$. Soit φ la substitution des α_i telle que $\tau = \varphi(\tau_x)$. Par hypothèse sur e et sur E, on sait que $e(x_r) = (a_0, e_0)$, et il existe E_0 tel que $S \models e_0 : E_0$ et $E_0 \vdash a_0 : \tau$. Comme $e(x_r)$ est une suspension, la seule évaluation possible est de la forme :

$$\frac{e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0/s \Rightarrow r}{e \vdash x_r/s \Rightarrow r}$$

On applique l'hypothèse de récurrence à l'expression a_0, de type τ, dans les environnements e_0, E_0, s, S. Il vient $r = v'/s'$ et il existe S' prolongeant S tel que

$$S' \models v' : \tau \quad \text{et} \quad \models s' : S'.$$

C'est le résultat attendu.

• **Cas du `let`.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x_r \mapsto \mathtt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \mathtt{let\ poly}\ x_r = a_1\ \mathtt{in}\ a_2 : \tau_2}$$

On a une seule possibilité d'évaluation :

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2/s \Rightarrow r}{e \vdash (\mathtt{let\ poly}\ x_r = a_1\ \mathtt{in}\ a_2)/s \Rightarrow r}$$

On montre $S \models (a_1, e) : \mathtt{Gen}(\tau_1, E)$. On a $\mathtt{Gen}(\tau_1, E) = \forall \alpha_1 \ldots \alpha_n . \tau_1$, et les α_i ne sont pas libres dans E. Soit φ une substitution sur les α_i. Par la proposition 1.2, on a $\varphi(E) \vdash a_1 : \varphi(\tau_1)$. Comme $\varphi(E) = E$ et $S \models e : E$ par hypothèse, on peut prendre E comme environnement requis par la définition de $\models$ sur les schémas. D'où $S \models (a_1, e) : \mathtt{Gen}(\tau_1, E)$. Notant

$$e_1 = e + x \mapsto v_1 \qquad\qquad E_1 = E + x \mapsto \mathtt{Gen}(\tau_1, E),$$

on a donc $S_1 \models e_1 : E_1$. On applique l'hypothèse de récurrence à a_2, e_1, E_1, s_1, S_1. Il vient que r est de la forme v_2/s_2, et il existe S_2 tel que

$$S_2 \models v_2 : \tau_2 \text{ et } \models s_2 : S_2 \text{ et } S_2 \text{ prolonge } S_1.$$

C'est bien le résultat attendu. □

6.3.2 Cas des canaux

Voici les relations de typage sémantique utilisées (Γ est un typage des canaux) :

$\Gamma \models v : \tau$	v est une valeur correcte du type τ
$\Gamma \models (a, e) : \sigma$	la suspension (a, e) est une valeur correcte de toutes les instances du schéma σ
$\Gamma \models e : E$	les valeurs et les suspensions contenues dans l'environnement d'évaluation e appartiennent bien aux schémas correspondants dans E
$\models u :? \Gamma$	les événements de réception (de la forme $c\,?\,v$) contenus dans la séquence d'événements u respectent les types des canaux attribués par Γ
$\models u :! \Gamma$	les événements d'émission (de la forme $c\,!\,v$) contenus dans la séquence d'événements u respectent les types des canaux attribués par Γ

Voici leur définition exacte :

- $\Gamma \models cst : \texttt{unit}$ si cst est `()`
- $\Gamma \models cst : \texttt{int}$ si cst est un entier
- $\Gamma \models cst : \texttt{bool}$ si cst est `true` ou `false`
- $\Gamma \models (v_1, v_2) : \tau_1 \times \tau_2$ si $\Gamma \models v_1 : \tau_1$ et $\Gamma \models v_2 : \tau_2$
- $\Gamma \models c : \tau\ \texttt{chan}$ si $c \in \mathrm{Dom}(\Gamma)$ et $\tau = \Gamma(c)$
- $\models (f_s, x_s, a, e) : \tau_1 \to \tau_2$ s'il existe un environnement de typage E tel que

$$\Gamma \models e : E \quad \text{et} \quad E \vdash (f_s\ \texttt{where}\ f_s(x_s) = a) : \tau_1 \to \tau_2$$

- $\Gamma \models (a, e) : \forall \alpha_1 \dots \alpha_n . \tau$ si pour toute substitution φ de domaine inclus dans $\{\alpha_1 \dots \alpha_n\}$, il existe un environnement de typage E tel que $\Gamma \models e : E$ et $E \vdash a : \varphi(\tau)$
- $\Gamma \models e : E$ si $\mathrm{Dom}(e) = \mathrm{Dom}(E)$, et pour tout $x_s \in \mathrm{Dom}(e)$, $E(x_s)$ est un type simple τ tel que $\Gamma \models e(x_s) : \tau$, et pour tout $x_r \in \mathrm{Dom}(e)$, $E(x_r)$ est un schéma σ tel que $\Gamma \models e(x_r) : \sigma$.
- $\models u :? \Gamma$ si pour tout événement de réception $c\,?\,v$ appartenant à la séquence u, on a $\Gamma \models v : \Gamma(c)$
- $\models u :! \Gamma$ si pour tout événement d'émission $c!v$ appartenant à la séquence u, on a $\Gamma \models v : \Gamma(c)$.

Comme dans la partie 3.3.2, on se donne un terme clos bien typé a_0, dont toutes les sous-expressions de la forme `newchan`(a) sont distinctes. On fixe une dérivation $\mathcal{T}$ de typage de a_0, et une dérivation $\mathcal{E}$ d'évaluation de a_0. On construit un typage des canaux Γ adapté à $\mathcal{T}$ et $\mathcal{E}$ comme expliqué dans la partie 3.3.2.

Proposition 6.2 (Sûreté forte pour les canaux) *Soit a une sous-expression de a_0. Soit $e \vdash a \stackrel{w}{\Longrightarrow} r$ la conclusion de la sous-dérivation de D_e décrivant l'évaluation de a, et $E \vdash a : \tau$ la conclusion de la sous-dérivation de D_t décrivant le typage de a. On suppose $\Gamma \models e : E$.*

1. *Si $\models w :? \Gamma$, alors $r \neq \texttt{err}$, au contraire r est une valeur v, qui vérifie $\Gamma \models v : \tau$, et de plus $\models w :! \Gamma$.*
2. *Si $w = w'.c\,!\,v.w''$ et $\models u' :? \Gamma$, alors $\Gamma \models v : \Gamma(c)$.*

Démonstration : la preuve procède par récurrence sur la taille de la sous-dérivation d'évaluation. On raisonne par cas sur a, et donc sur la dernière règle utilisée dans la dérivation de typage. La plupart des cas sont exactement les mêmes que dans la preuve de la proposition 3.9. Je donne seulement les cas qui diffèrent.

• **Cas d'une variable stricte.**

$$\frac{\tau \leq E(x_s)}{E \vdash x_s : \tau}$$

On sait par le typage que $x \in \mathrm{Dom}(E) = \mathrm{Dom}(e)$, donc la seule évaluation possible est $e \vdash x_s \stackrel{\varepsilon}{\Longrightarrow} e(x_s)$. Par hypothèse sur e et E, on a $\Gamma \models e(x_s) : E(x_s)$. Comme $E(x_s)$ est un type simple, on a $\tau = E(x_s)$. D'où (1). (2) est trivialement vraie, parce que nécessairement $w = \varepsilon$.

• **Cas d'une variable retardée.**

$$\frac{\tau \leq E(x_r)}{E \vdash x_r : \tau}$$

Écrivons $E(x_r) = \forall \alpha_1 \ldots \alpha_n.\, \tau_x$. Soit φ la substitution sur les α_i telle que $\tau = \varphi(\tau_x)$. Par hypothèse sur e et sur E, on sait que $e(x_s) = (a_0, e_0)$, et il existe E_0 tel que $S \models e_0 : E_0$ et $E_0 \vdash a_0 : \tau$. Comme $e(x_r)$ est une suspension, la seule évaluation possible est de la forme :

$$\frac{e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0 \stackrel{w}{\Longrightarrow} r}{e \vdash x_r \stackrel{w}{\Longrightarrow} r}$$

On applique l'hypothèse de récurrence à l'expression a_0, de type τ, dans les environnements e_0, E_0, et sous la séquence d'événements w. On obtient les propriétés (1) et (2) pour l'évaluation de a_0, d'où (1) et (2) pour l'évaluation de x_r.

• **Cas du `let`.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x_r \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let poly } x_r = a_1 \texttt{ in } a_2 : \tau_2}$$

On a une seule possibilité d'évaluation :

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2 \stackrel{w}{\Longrightarrow} r}{e \vdash \texttt{let poly } x_r = a_1 \texttt{ in } a_2 \stackrel{w}{\Longrightarrow} r}$$

On montre $\Gamma \models (a_1, e) : \texttt{Gen}(\tau_1, E)$ comme dans la preuve de la proposition 6.1. Notant

$$e_1 = e + x \mapsto v_1 \qquad\qquad E_1 = E + x \mapsto \texttt{Gen}(\tau_1, E),$$

on a donc $\Gamma \models e_1 : E_1$. On applique l'hypothèse de récurrence à a_2, e_1, E_1 et w. Il vient les propriétés (1) et (2) attendues. □

6.3.3 Cas des continuations

On emploie les relations de typage sémantique que voici :

$\models v : \tau$	v est une valeur correcte du type τ
$\models (a, e) : \sigma$	la suspension (a, e) est une valeur correcte de toutes les instances du schéma σ
$\models e : E$	les valeurs contenues dans l'environnement d'évaluation e appartiennent bien aux schémas correspondants dans E
$\models k :: \tau$	la continuation k accepte toutes les valeurs du type τ

Voici leur définition exacte :

- $\models cst : \texttt{unit}$ si cst est `()`
- $\models cst : \texttt{int}$ si cst est un entier
- $\models cst : \texttt{bool}$ si cst est `true` ou `false`
- $\models (v_1, v_2) : \tau_1 \times \tau_2$ si $\models v_1 : \tau_1$ et $\models v_2 : \tau_2$
- $\models k : \tau$ `cont` si $\models k :: \tau$
- $\models (f_s, x_s, a, e) : \tau_1 \to \tau_2$ s'il existe un environnement de typage E tel que :

$$\models e : E \quad \text{et} \quad E \vdash (f_s \texttt{ where } f_s(x_s) = a) : \tau_1 \to \tau_2$$

- $\models (a, e) : \forall \alpha_1 \ldots \alpha_n . \tau$ si pour toute substitution φ de domaine inclus dans $\{\alpha_1 \ldots \alpha_n\}$, il existe un environnement de typage E tel que $\models e : E$ et $E \vdash a : \varphi(\tau)$
- $\models e : E$ si $\mathrm{Dom}(e) = \mathrm{Dom}(E)$, et pour tout $x_s \in \mathrm{Dom}(e)$, $E(x_s)$ est un type simple τ tel que $\models e(x_s) : \tau$, et pour tout $x_r \in \mathrm{Dom}(e)$, $E(x_r)$ est un schéma σ tel que $\models e(x_r) : \sigma$.
- $\models \texttt{stop} :: \tau$ pour tout type τ
- $\models \texttt{app1c}(a, e, k) :: \tau_1 \to \tau_2$ s'il existe un environnement de typage E tel que

$$E \vdash a : \tau_1 \quad \text{et} \quad \models e : E \quad \text{et} \quad \models k :: \tau_2$$

- $\models \texttt{app2c}(f_s, x_s, a, e, k) :: \tau$ s'il existe un environnement de typage E et un type τ' tels que

$$E \vdash (f_s \texttt{ where } f_s(x_s) = a) : \tau \to \tau' \quad \text{et} \quad \models e : E \quad \text{et} \quad \models k :: \tau'$$

- $\models \texttt{pair1c}(a, e, k) :: \tau$ s'il existe un environnement de typage E et un type τ' tels que

$$E \vdash a : \tau' \quad \text{et} \quad \models e : E \quad \text{et} \quad \models k :: \tau \times \tau'$$

- $\models \texttt{pair2c}(v, k) :: \tau$ s'il existe un type τ' tel que

$$\models v : \tau' \quad \text{et} \quad \models k :: \tau' \times \tau$$

- $\models \texttt{primc}(\texttt{callcc}, k) :: \tau\ \texttt{cont} \to \tau$ si $\models k :: \tau$
- $\models \texttt{primc}(\texttt{throw}, k) :: \tau\ \texttt{cont} \times \tau$ pour tout τ.

Proposition 6.3 (Sûreté faible pour les continuations)

1. *Soient a une expression, τ un type, e un environnement d'évaluation, E un environnement de typage, k une continuation et r une réponse tels que*

$$E \vdash a : \tau \quad et \quad \models e : E \quad et \quad \models k :: \tau \quad et \quad e \vdash a; k \Rightarrow r.$$

   Alors $r \neq \texttt{err}$.

2. *Soient v une valeur, k une continuation, τ un type et r une réponse tels que*

$$\models v : \tau \quad et \quad \models k :: \tau \quad et \quad \vdash v \triangleright k \Rightarrow r.$$

   Alors $r \neq \texttt{err}$.

Démonstration : la preuve est une récurrence sur la taille de la dérivation d'évaluation. On raisonne, pour (1), par cas sur a et donc sur la dernière règle utilisée dans la dérivation de typage, et pour (2) par cas sur k. Je donne seulement les cas qui ne sont pas les mêmes que dans la preuve de la proposition 3.12.

• (1), cas d'une variable stricte.

$$\frac{\tau \leq E(x_s)}{E \vdash x_s : \tau}$$

On sait par le typage que x_s appartient au domaine de E, et par l'hypothèse $\models e : E$ on a $\mathrm{Dom}(E) = \mathrm{Dom}(e)$. Donc la seule évaluation possible est :

$$\frac{x_s \in \mathrm{Dom}(e) \qquad \vdash e(x_s) \triangleright k \Rightarrow r}{e \vdash cst; k \Rightarrow r}$$

On sait par hypothèse que $\models e(x_s) : E(x_s)$, et $E(x_s) = \tau$ puisque $E(x_s)$ est un type simple. D'où $\models e(x_s) : \tau$. Appliquant l'hypothèse de récurrence (2) à l'évaluation $\vdash e(x_s) \triangleright k \Rightarrow r$, il vient $r \neq \texttt{err}$.

• (1), cas d'une variable retardée.

$$\frac{\tau \leq E(x_r)}{E \vdash x_r : \tau}$$

Écrivons $E(x_r) = \forall\alpha_1 \ldots \alpha_n.\, \tau_x$. Soit φ la substitution sur les α_i telle que $\tau = \varphi(\tau_x)$. Par hypothèse sur e et sur E, on sait que $e(x_s) = (a_0, e_0)$, et il existe E_0 tel que $\models e_0 : E_0$ et $E_0 \vdash a_0 : \tau$. Comme $e(x_r)$ est une suspension, la seule évaluation possible est de la forme :

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0; k \Rightarrow r}{e \vdash x_r; k \Rightarrow r}$$

On applique l'hypothèse de récurrence (1) à $a_0; k$ dans E_0 et e_0, il s'ensuit le résultat attendu : $r \neq \texttt{err}$.

• **(1), cas du `let`.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x_r \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let poly } x_r = a_1 \texttt{ in } a_2 : \tau_2}$$

On a une seule possibilité d'évaluation :

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2; k \Rightarrow r}{e \vdash (\texttt{let poly } x_r = a_1 \texttt{ in } a_2); k \Rightarrow r}$$

On montre $\models (a_1, e) : \texttt{Gen}(\tau_1, E)$ comme dans la preuve de la proposition 6.1. Notant

$$e_1 = e + x_r \mapsto (a_1, e) \qquad\qquad E_1 = E + x_r \mapsto \texttt{Gen}(\tau_1, E),$$

on a donc $\models e_1 : E_1$. On applique l'hypothèse de récurrence (1) à $a_2; k$, e_1, et E_1. Il vient $r \neq \texttt{err}$, comme annoncé. □

6.4 Discussion

L'adoption du polymorphisme par nom permet de typer correctement les références, les canaux et les continuations tout en conservant les règles de typage et l'algèbre de types de ML, qui sont simples et faciles à comprendre — au contraire des systèmes de types pour la sémantique stricte, qui nécessitent une algèbre de types plus complexes et des règles de typages plus difficiles. En particulier, l'écriture manuelle de types dans les interfaces de modules reste facile, alors qu'elle est difficile dans les systèmes des chapitres 3 et 4, ou dans les systèmes d'effets les plus évolués qu'on a présenté au chapitre 5. MLN, la variante de ML avec polymorphisme par nom étudiée ici, semble donc être une alternative intéressante à ML originel, dès lors qu'on envisage des traits non purement applicatifs.

Cette variante est cependant moins puissante que ML sur un point : on ne peut pas y définir un objet polymorphe une fois pour toute, et voir sa valeur partagée entre toutes les utilisations de l'objet. En conséquence, certains programmes n'ont plus la même sémantique ; d'autres, plus la même efficacité. On va maintenant discuter de la portée pratique de ces différences.

6.4.1 Différences sémantiques

Comme exemples de programmes qui n'ont pas la même sémantique dans les deux langages, on trouve tous les cas où le calcul d'un objet polymorphe a des effets de bords : les effets de bords sont effectués une seule fois, au moment de la création, en ML, mais zéro, une ou plusieurs fois en MLN (une fois pour chaque utilisation). Exemple : en ML, l'évaluation de

```
let f = print("Hi!"); λx.x in f(f(f))
```

affiche "`Hi !`" une seule fois, alors que la phrase correspondante en MLN,

```
let poly f = print("Hi!"); λx.x in f(f(f))
```

affiche "`Hi !`" trois fois. Voici un autre exemple, qui suppose défini un générateur de marques `gensym` :

```
let stamper =
  let stamp = gensym() in λx.(x, stamp)
in ...
```

En ML, la fonction `stamper` ainsi définie prend un objet quelconque et l'apparie avec la marque préalablement obtenue — toujours la même. La traduction directe en MLN se comporte différemment :

```
let poly stamper =
  let val stamp = gensym() in λx.(x, stamp)
in ...
```

Chaque appel de `stamper` provoque la ré-évaluation de `gensym()`, et donc une marque différente est attribuée à chaque fois. Pour retrouver le comportement du programme d'origine, il faut restructurer le programme, et écrire :

```
let val stamp = gensym() in
let poly stamper = λx.(x, stamp) in ...
```

Dans des exemples plus compliqués et encore plus artificiels, des transformations de plus grande envergure peuvent se révéler nécessaire. Mon expérience pratique est que la situation illustrée ci-dessus (définir un objet polymorphe par une expression avec effets de bords) est très rare. J'ai fait passer environ 10000 lignes de programmes ML dans un compilateur MLN expérimental (voir ci-après, partie 6.4.3), sans rencontrer une seule fois cette situation. Pour traduire les programmes, je n'ai eu qu'à remplacer les `let` par des `let val` ou des `let poly` : aucune modification de plus grande ampleur (comme dans l'exemple `stamper`) ne s'est révélée nécessaire.

Contexte. Un langage avec effets de bords et une construction non stricte comme le `let poly` peut apparaître comme propice à certaines erreurs de programmation : celles où un effet de bord ne se produit pas au moment attendu. On dit souvent que la sémantique la plus intuitive pour un langage avec effets de bord est la sémantique stricte avec évaluation de gauche à droite, car c'est celle où le séquencement des effets de bords suit au plus près l'ordre du texte source. Certains langages impératifs ne suivent cependant pas ce précepte. Un précédent fameux est Algol 60, qui

utilise l'appel par nom ; une étude récente (le langage Forsythe de Reynolds [83]) reprend cette particularité. Le langage Caml mélange effets de bord et évaluation paresseuse avec partage des évaluations [98, chap. 4], rendant très difficile la prévision du moment où un effet de bord dans une expression retardée se produit. Il y a aussi, en Caml, la construction a `where` $x = b$, où l'expression b est évaluée avant l'expression a, bien qu'elle apparaisse après dans le texte du programme. Enfin, des langages comme C, C++, Modula-3 ne spécifient pas l'ordre d'évaluation des sous-expressions d'une expression arithmétique. L'expérience avec C montre que cette non-spécification mène parfois à des erreurs coriaces, mais a généralement un effet bénéfique sur la lisibilité des programmes : elle force les programmeurs à mettre plus en évidence les effets de bords et l'ordre dans lequel ils doivent se produire. L'adoption d'une sémantique non-stricte pour le polymorphisme devrait avoir, à mon avis, des conséquences pratiques très similaires. □

6.4.2 Différences d'efficacité

Les deux langages ML et MLN sont équivalents lorsqu'on les restreint à des programmes où tous les objets polymorphes sont définis par des expressions dont l'évaluation termine toujours, et ne produit pas d'effets de bord observables. Dans ce cas, des différences d'efficacité peuvent cependant apparaître : en MLN, les objets polymorphes ne sont pas partagés, mais recalculés à chaque utilisation ; on peut donc craindre une exécution moins rapide qu'en ML. Voyons ce qu'il en est en pratique.

Dans la plupart des programmes, les seuls objets polymorphes sont des définitions de fonctions $\lambda x.\, a$. Leur évaluation se réduit à la construction d'une fermeture, ce qui se fait en temps constant (et assez faible) ; reconstruire la fermeture à chaque utilisation de l'identificateur polymorphe n'est donc pas très coûteux, et en tout cas ne change pas le comportement en temps du programme. De plus, cette construction de fermeture peut être évitée dans de nombreux cas, en utilisant les techniques habituelles de dé-curryfication : la transformation, au cours de la compilation, de fonctions curryfiées en fonctions à plusieurs arguments [2, partie 6.2]. Voici un fragment de programme qui revient souvent :

```
let poly f = λx. a in
    ... f(1) ... f(true) ...
```

Avec la sémantique du polymorphisme par nom, ce programme est exactement compilé comme le programme ML suivant :

```
let f = λ(). λx. a in
    ... f()(1) ... f()(true) ...
```

Les techniques de dé-curryfication transforment `f` en une fonction à deux arguments, `()` et `x` ; les deux applications de `f` sont transformés en deux appels directs à `f`, presque aussi efficaces que les applications simples `f(1)` et `f(true)`. Le seul surcoût éventuel est qu'il faut passer `()` comme argument supplémentaire ; ceci ne coûte rien avec certaines techniques de représentation des données [50].

Certains cas d'application partielle de fonctions curryfiées peuvent cependant créer des objets polymorphes qui coûtent cher à recalculer. Une fonction curryfiée peut effectuer des calculs entre la

réception de son premier et de son deuxième argument. Lier par un `let` le résultat d'une application partielle peut donc conduire à factoriser ces calculs. Avec le polymorphisme par nom, ce partage est perdu si le résultat de l'application partielle doit rester polymorphe. Voici un exemple presque réaliste où cette situation se produit ; c'est le seul que j'ai pu trouver.

Exemple. On considère une fonction qui trie des couples (clé, donnée) par ordre croissant des clés. Supposons que les clés et les données associées ne sont pas fournies ensembles, sous forme d'un tableau de paires par exemple, mais séparément, sous forme d'un tableau de clés et d'un tableau de données. Le résultat de la fonction est le tableau des données, trié. Pour tirer parti de l'application partielle, la manière astucieuse d'écrire cette fonction est de calculer la permutation de tri (un tableau d'entiers) dès que le tableau des clés est connu, et de renvoyer une fonction qui se contente d'appliquer la permutation sur le tableau de données fourni :

```
let weird_sort =
  λorder. λkeys.
    let permut = ... in λitems. apply_permut(permut)(items)
```

Cette écriture se révèle la plus efficace dans le cas où on doit trier plusieurs tableaux de données suivant le même tableau de clés :

```
let f = weird_sort (prefix <) (lots_of_integers) in
    ... f(lots_of_strings) ... f(lots_of_booleans) ...
```

Dans la phrase ci-dessus, la fonction intermédiaire `f` est polymorphe (de type $\forall\alpha.\ \alpha$ `array` $\rightarrow$ α `array`), et peut donc être appliquée à des tableaux de différents types — sans avoir à trier à nouveau le tableau de clés. Ce dernier point vaut en ML, mais pas en MLN : pour que `f` soit polymorphe, il faut le lier par un `let poly` ; dès lors, chaque application `f(t)` s'évalue comme

```
weird_sort (prefix <) (lots_of_integers) (t)
```

On a donc perdu tous les bénéfices de l'application partielle. Sur un tel exemple, MLN peut se révéler nettement moins efficace que ML. □

6.4.3 Une mise en pratique

Pour tenter d'estimer l'impact pratique des différences d'efficacité présentées ci-dessus, j'ai réalisé un prototype de compilateur ML avec polymorphisme par nom à partir de mon système Caml Light. J'ai ensuite comparé l'efficacité du code produit par ce prototype avec celle du code produit par Caml Light d'origine, qui, lui, implémente la sémantique du polymorphisme par valeur.

6.4.3.1 Le modèle d'exécution de Caml Light

Avant de commenter les résultats obtenus, je vais décrire brièvement le modèle d'exécution du système Caml Light, et son adaptation au polymorphisme par nom. (Pour plus de détails, on se reportera au chapitre 3 de [49].) Le modèle d'exécution de Caml Light se distingue par un traitement particulier des applications multiples, dont le but est d'effectuer la dé-curryfication "au vol". Je vais tout d'abord montrer la compilation du noyau fonctionnel de ML avec polymorphisme par valeur, puis montrer comment ces techniques s'adaptent au polymorphisme par nom.

La compilation suit deux schémas : le premier, $CT(a)$, s'applique aux expressions a en position d'appel terminal ; l'autre, $CN(a)$, s'applique aux expressions en position non terminale. Une variable se compile par une instruction d'accès dans l'environnement. (La structure de l'environnement est laissée abstraite dans cette discussion.)

$$CT(x) = \texttt{Access}(x) \qquad CN(x) = \texttt{Access}(x)$$

Une application de fonction à n arguments (curryfiés) se compile comme suit :

$$\begin{aligned} CT(a(a_1)\ldots(a_n)) &= CN(a_n);\texttt{Push};\ldots;CN(a_1);\texttt{Push};CN(a);\texttt{Appterm} \\ CN(a(a_1)\ldots(a_n)) &= \texttt{Pushmark};CN(a_n);\texttt{Push};\ldots;CN(a_1);\texttt{Push};CN(a);\texttt{Apply} \end{aligned}$$

Pour une application non terminale, on empile une valeur distinguée, la "marque", qui sépare les arguments fournis à a des autres valeurs qui se trouvent sur la pile. Puis on évalue les arguments en commençant par le dernier, et on empile leurs valeurs. On évalue ensuite l'expression en position fonctionnelle en une fermeture, et on saute à la partie code de cette fermeture. Pour une application terminale, il est inutile d'empiler la marque : les arguments fournis à a s'ajoutent aux arguments restants sur la pile.

$$\begin{aligned} CT(\lambda x.\, a) &= \texttt{Grab}(x);CT(a) \\ CN(\lambda x.\, a) &= \texttt{Clos}(CT(\lambda x.\, a);\texttt{Return}) \end{aligned}$$

Une abstraction sur x en position terminale se traduit par l'insertion d'une instruction $\texttt{Grab}(x)$. (Pour rester simple, je ne traite pas les fonctions récursives dans cette discussion.) À l'exécution, cette instruction teste et dépile le sommet de la pile d'arguments. Si le sommet est égal à la marque, cela signifie qu'il n'y a plus d'arguments disponibles pour la fonction. L'instruction $\texttt{Grab}(x)$ construit alors une fermeture du code $\texttt{Grab}(x);CT(a)$ par l'environnement courant, et la retourne à l'appelant. Si le sommet de la pile n'est pas une marque, c'est l'argument auquel l'abstraction est appliquée. L'instruction $\texttt{Grab}(x)$ lie cette valeur à l'identificateur x dans l'environnement courant, et poursuit l'évaluation de a (le corps de la lambda-abstraction) en séquence.

Une abstraction en position non terminale n'est jamais immédiatement appliquée, et donc on compile une instruction `Clos`, qui construit une fermeture du code pour $\lambda x.\, a$ avec l'environnement courant. L'instruction `Return`, qui termine le code de la fonction, se comporte symétriquement de `Grab` : si le sommet de la pile est une marque, tous les arguments ont été consommés, et on retourne la valeur calculée à l'appelant ; si le sommet de la pile n'est pas une marque, il reste des arguments à consommer, et donc `Return` applique (par un appel terminal) la valeur calculée (une fermeture, nécessairement) aux arguments restants.

$$\begin{aligned} CT(\texttt{let}\ x = a_1\ \texttt{in}\ a_2) &= CN(a_1);\texttt{Bind}(x);CT(a_2) \\ CN(\texttt{let}\ x = a_1\ \texttt{in}\ a_2) &= CN(a_1);\texttt{Bind}(x);CN(a_2);\texttt{Unbind}(x) \end{aligned}$$

La liaison `let` se compile très classiquement en les deux instructions `Bind` et `Unbind`, qui ajoutent ou enlèvent une liaison dans l'environnement, sans tester la marque comme le fait `Grab`.

Le point fort de ce mécanisme d'exécution est son bon comportement vis-à-vis des fonctions curryfiées. Par exemple, le fragment de code suivant s'exécute sans construction de fermetures intermédiaires, et avec un seul aller-retour entre l'appelant et l'appelé.

```
let f = λx. λy. λz. x + y + z
in f(2)(3)(4)
```

Dans un modèle classique, comme par exemple la machine SECD [46], on aurait construit deux fermetures intermédiaires (correspondant aux applications partielles `f(1)` et `f(1)(2)`), et trois aller-retours entre l'appelant et l'appelé se seraient produits. Le modèle d'exécution de Caml Light atteint donc presque la même efficacité que les techniques de décurryfication statique habituelles : les trois arguments de l'application curryfiée sont passés d'un seul coup à la fonction. (La seule perte d'efficacité est due aux tests sur le sommet de pile effectués par `Grab`.) Ce modèle se révèle plus puissant que la décurryfication statique dans le cas de fonctions curryfiées qui entremêlent passage d'arguments et calculs internes :

```
let f = λx. let u = fib(x) in λy. let v = fact(y) in λz. u + v + z
in f(2)(3)(4)
```

Dans cet exemple, les trois arguments sont toujours passés en un seul coup, et on ne fait toujours qu'un aller-retour. Les techniques de décurryfication statique ne s'appliquent généralement pas à des situations complexes comme celle-ci.

6.4.3.2 Adaptation au polymorphisme par nom

Le modèle d'exécution résumé ci-dessus sépare clairement suspension de l'évaluation (instruction `Clos`) et passage d'arguments (instruction `Grab`). Il permet donc la définition facile de suspensions, et par conséquent s'adapte bien au polymorphisme par nom.

Le `let poly` se compile naturellement par la création d'une suspension et son ajout dans l'environnement.

$$\begin{aligned} CT(\texttt{let poly } x_r = a_1 \texttt{ in } a_2) &= \texttt{Clos}(CT(a_1); \texttt{Return}); \texttt{Bind}(x_r); CT(a_2) \\ CN(\texttt{let poly } x_r = a_1 \texttt{ in } a_2) &= \texttt{Clos}(CT(a_1); \texttt{Return}); \texttt{Bind}(x_r); CN(a_2); \texttt{Unbind}(x_r) \end{aligned}$$

L'accès à une variable liée par un `let poly` se traduit par une application à zéro arguments.

$$\begin{aligned} CT(x_r) &= \texttt{Access}(x_r); \texttt{Appterm} \\ CN(x_r) &= \texttt{Pushmark}; \texttt{Access}(x_r); \texttt{Appterm} \end{aligned}$$

Si cette variable est immédiatement appliquée, ce qui est fréquent, on peut combiner en une seule application l'évaluation de la suspension avec l'application de la valeur de la suspension :

$$\begin{aligned} CT(x_r(a_1)\ldots(a_n)) &= CN(a_n); \texttt{Push}; \ldots; CN(a_1); \texttt{Push}; \texttt{Access}(x_r); \texttt{Appterm} \\ CN(x_r(a_1)\ldots(a_n)) &= \texttt{Pushmark}; CN(a_n); \texttt{Push}; \ldots; CN(a_1); \texttt{Push}; \texttt{Access}(x_r); \texttt{Apply} \end{aligned}$$

La première ligne se déduit des règles générales pour l'accès à une variable retardée et pour l'application terminale. La deuxième ligne ne se déduit pas des règles générales ; c'est une optimisation spéciale qui évite d'empiler deux marques, et donc de stopper une fois de trop l'évaluation de la suspension.

En conséquence, l'application d'une fonction polymorphe s'effectue aussi efficacement en polymorphisme par nom qu'en polymorphisme par valeur. Considérons le fragment de code typique :

Test	Sémantique par valeur	Sémantique par nom	Ralentissement nom/valeur	Quantité de polymorphisme
Fibonacci	5,9 s	6,3 s	6 %	nulle
Entiers de Church	2,5 s	2,9 s	16 %	élevée
Sieve	3,2 s	3,4 s	6 %	modérée
Word count	6,4 s	6,7 s	4 %	nulle
Boyer	16,0 s	18,0 s	12 %	faible
Knuth-Bendix	7,9 s	8,3 s	5 %	modérée
Générateur de lexers	2,1 s	2,3 s	9 %	faible
Compilateur Caml Light	7,1 s	8,3 s	16 %	faible

Figure 6.1: Comparaison entre polymorphisme par valeur/polymorphisme par nom

```
let poly f = λx. a in
    ... f(1) ...
    ... f(true) ...
```

Le code produit est de la forme suivante :

```
Clos(Grab(x); CT(a)); Bind(f);
    ...; Pushmark; Const(1); Push; Access(f); Apply; ...
    ...; Pushmark; Const(true); Push; Access(f); Apply; ...
```

C'est, à l'instruction près, le code que produit le schéma de compilation pour le polymorphisme par valeur sur le programme ML équivalent :

```
let f = λx. a in
    ... f(1) ...
    ... f(true) ...
```

Pour ce cas très fréquent, la sémantique par nom du polymorphisme n'introduit donc aucune inefficacité par-rapport à la sémantique par valeur.

6.4.3.3 Résultats expérimentaux

La figure 6.1 donne des temps d'exécution comparés entre Caml Light d'origine et Caml Light avec polymorphisme par nom. Les programmes de test se répartissent en : des programmes-jouets bien connus (la suite de Fibonacci, des manipulations d'entiers de Church, le crible d'Eratosthène sur listes, un compteur de mots dans un fichier) ; deux programmes de calcul symbolique de quelques centaines de lignes, le mini-démonstrateur de Boyer et une implémentation de l'algorithme de complétion de Knuth-Bendix ; et deux morceaux du système Caml Light réécrits en Caml Light avec polymorphisme par nom et "bootstrappés", le générateur d'analyseurs lexicaux (1000 lignes) et le compilateur lui-même (8000 lignes). Certains de ces programmes sont entièrement monomorphes (Fibonacci, word count). D'autres emploient des fonctions polymorphes à haute dose (entiers de Church). Les programmes plus réalistes manipulent pour l'essentiel des structures monomorphes, mais ils emploient souvent des fonctions polymorphes sur les listes, les tables de hachage, etc.

Sur les temps d'exécution, on voit que tous les programmes, y compris les programmes complètement monomorphes, sont ralentis lorsqu'on passe en polymorphisme par nom. L'explication est la suivante : une petite optimisation du compilateur ne s'applique plus lorsqu'on passe au schéma de compilation avec polymorphisme par nom de la partie précédente. Sans polymorphisme par nom, toutes les fonctions sont toujours appliquées à au moins un argument ; aussi, le `Grab` initial au début de chaque fonction est omis, et ce sont les instructions `Apply` et `Appterm` qui placent d'office le premier argument en tête de l'environnement. Avec polymorphisme par nom, ce n'est plus le cas, et il faut laisser le `Grab` initial au début de chaque fonction. Il en résulte un ralentissement d'environ 5%, à en juger par les exemples complètement monomorphes (Fibonacci et word count).

En plus de ce ralentissement général d'environ 5%, on observe sur les programmes utilisant le polymorphisme un ralentissement de 1 à 10%, qui représente le coût propre lié au changement de sémantique du polymorphisme. Il ne se dégage pas de corrélation nette entre la quantité de fonctions polymorphes dans le texte source et le ralentissement observé. Un programme assez monomorphe comme le compilateur est plus ralenti qu'un programme comportant nettement plus de fonctionnelles polymorphes comme l'implémentation de l'algorithme de Knuth-Bendix. Il est vrai que j'ai seulement estimé la proportion statique de fonctions polymorphes (combien d'appels de fonctions polymorphes figurent dans le source), et non pas la proportion dynamique (combien d'appels de fonctions polymorphes sont effectués à l'exécution), que je ne sais pas mesurer dans l'état actuel du système Caml Light.

Ces premiers résultats expérimentaux sont encourageants : ils montrent qu'un modèle d'exécution initialement conçu pour le polymorphisme par valeur peut s'adapter sans grands efforts au polymorphisme par nom, sans perte majeure d'efficacité. Je suis persuadé qu'ils s'étendent sans grandes différences de Caml Light à d'autres implémentations de ML munies de bons mécanismes de décurryfication. Ces résultats suffisent à réfuter l'objection a priori selon laquelle le polymorphisme par nom est inenvisageable, parce qu'essentiellement inefficace [29].

Conclusion

Arrivé au terme de ce travail, le moins que je puisse dire est que le typage polymorphe à la manière de ML ne s'étend pas naturellement d'un langage purement applicatif à un langage muni de structures de données modifiables, de canaux de communication en tant que valeurs, ou d'objets continuations.

Cette étude a fait apparaître trois propriétés désirables d'une telle extension qui, empiriquement, semblent très difficiles à concilier. Premièrement, le système de types doit être expressif. En particulier, il doit faire bénéficier les traits non applicatifs de toute la puissance du typage polymorphe. Deuxièmement, l'algèbre de types doit rester suffisamment simple pour ne pas s'opposer à la décomposition des programmes en modules. En particulier, la spécification du type d'une fonction doit indiquer ce que la fonction calcule, mais ne doit pas imposer comment elle le calcule. Troisièmement, on souhaite conserver la même sémantique pour le polymorphisme qu'en ML, c'est-à-dire la sémantique stricte. Cette sémantique apparaît comme la plus intuitive pour un langage à polymorphisme implicite ; c'est aussi la plus efficace à l'exécution.

Les approches connues satisfont deux de ces impératifs, mais pas le troisième. L'approche "historique", celle du standard ML, préserve la sémantique stricte du polymorphisme et (jusqu'à un certain point) la compatibilité avec la programmation modulaire, mais sacrifie l'expressivité du typage, en le rendant incapable d'attribuer des types complètement polymorphes à de nombreuses fonctions génériques. Les systèmes à base de variables dangereuses et de typage des fermetures présentés dans cette Thèse, ainsi que les systèmes d'effets les plus récents, se révèlent très expressifs, et conservent la sémantique stricte du polymorphisme ; cependant, des difficultés avec la programmation modulaire sont à craindre, car les types des fonctions y sont beaucoup plus précis qu'en ML, et révèlent davantage la manière dont une fonction est implémentée. Enfin, le polymorphisme par nom, au sens du chapitre 6, est tout à fait satisfaisant du point de vue de l'expressivité et de la compatibilité avec les modules, mais attribue au polymorphisme une sémantique différente de celle de ML, et vraisemblablement moins intuitive pour le programmeur que celle de ML.

De même qu'un tabouret à trois pieds dont un trop court est stable mais inconfortable, de même chacune des trois solutions partielles énumérées ci-dessus peut se défendre pour une large classe d'utilisations de ML, mais n'est pas satisfaisante en général. Sacrifier l'expressivité du typage polymorphe des constructions non applicatives convient pour un style de programmation "essentiellement applicatif", où les traits non applicatifs sont utilisés rarement et de façon très ponctuelle. Compliquer la spécification des types convient pour l'écriture rapide de programmes de taille modeste, où la décomposition modulaire n'a pas besoin d'être explicitée complètement.

Adopter la sémantique du polymorphisme par nom, enfin, convient si on ne cherche pas à tout prix la compatibilité avec ML, et si on ne craint pas des pertes d'efficacité dans certaines situations. Il n'empêche que le problème initial reste ouvert, de proposer une extension naturelle et pleinement satisfaisante du système de types de ML à certains traits importants des langages algorithmiques.

Dans la présente Thèse, j'ai néanmoins apporté à ce problème deux contributions d'une certaine importance. Première contribution : le système de types avec variables dangereuses et typage des fermetures. Dans le cadre de la sémantique stricte du polymorphisme, ce système est, à ma connaissance et à l'heure où j'écris ces lignes, le plus expressif de tous les systèmes de types polymorphes connus pour les références ou les continuations. L'important n'est pas tant cette grande expressivité — les systèmes d'effets avec régions et masquage d'effets semblent faire jeu égal avec mon système dans les situations pratiques — que la manière dont elle est obtenue. Ce système repose exclusivement sur le fait que les types décrivent ce qu'il y a dans les valeurs. Certes, ce n'est pas vrai dans les systèmes de types classiques, et c'est pourquoi il a fallu ajouter le typage des fermetures ; mais, même enrichis de types de fermetures, *les types n'expriment toujours que des propriétés statiques des expressions* ("voici une approximation de ce que calcule cette expression"), et non pas des propriétés dynamiques des expressions ("voici une approximation de la manière dont cette expression calcule"). Les approches rivales, des variables impératives du standard ML aux systèmes d'effets les plus modernes, font toutes passer dans les types des informations sur le comportement dynamique des expressions. Cette pratique me semble contraire à une certaine idée du typage ; en particulier, elle mène tout droit à la sur-spécification par les types dans les interfaces de modules. Je suis donc heureux d'avoir démontré par l'exemple que, contrairement à ce qu'on pourrait croire, il n'est pas nécessaire de refléter dans les types le comportement dynamique des expressions pour garder le contrôle des références, des canaux et des continuations polymorphes. (Ma joie est tempérée par le fait que cela ne suffit malheureusement pas pour éviter l'écueil de la sur-spécification.)

Deuxième contribution de cette Thèse : proposer clairement le polymorphisme par nom comme une variante de ML très intéressante en présence de traits non purement applicatifs. Je ne prétends pas être le premier à remarquer que les références et les continuations polymorphes ne mettent pas en danger la sûreté du typage lorsque le polymorphisme est interprété suivant la sémantique par nom. Cette idée est dans l'air depuis un certain temps. Le seul mérite que je m'attribue est d'avoir écrit les preuves de sûreté, confirmant ainsi l'intuition, et d'avoir montré qu'en pratique le polymorphisme par nom peut fort bien s'intégrer dans un langage à la ML. Oui, la sémantique du polymorphisme par nom est compatible avec la syntaxe implicite et avec l'inférence de types ; nul n'est besoin de passer à un langage avec polymorphisme explicite pour bénéficier de la sémantique par nom. Non, le polymorphisme par nom n'est pas d'une inefficacité rédhibitoire : bien compilé, son surcoût par-rapport au polymorphisme par valeur est nul dans les cas courants. Certes, le langage qu'on obtient n'est plus vraiment ML ; mais c'est une solution très raisonnable et tout à fait utilisable en pratique au problème de typer un langage algorithmique avec des types polymorphes.

Faisons maintenant le point sur les techniques employées pour atteindre ces résultats. Tout d'abord, j'ai pu raisonner sur trois traits difficiles à décrire, car fortement "non-fonctionnels" et d'une grande expressivité, en n'employant que des méthodes complètement élémentaires : comme structures, des algèbres de termes ; comme méthode de définition, des règles d'inférences ; comme technique de preuve, la récurrence structurelle. On y perd peut-être en élégance par-rapport à

d'autres formalismes mathématiquement plus riches ; mais on y gagne des preuves accessibles et qui s'adaptent bien à trois situations assez diverses. De même que la sémantique relationnelle se révèle très proche d'une exécution directe par machine, de même les preuves que j'ai données semblent très proches d'une vérification directe par machine.

Les méthodes qui mènent à ce résultat sont simples : introduire des indirections pour décrire par des termes finis des situations cycliques (états mémoires, pour les références ; ensembles de contraintes, pour les types de fermetures) ; raisonner par récurrence structurelle sur les valeurs ; et considérer les fermetures et les continuations comme des expressions en attente, dont le typage sémantique se confond avec le typage syntaxique, et non pas comme des transformateurs de valeurs, dont le typage sémantique se fait par la condition de continuité. Ces techniques se trouvent pour la plupart dans la thèse de Tofte [91] ; mais je pense les avoir portées plus loin, en particulier en supprimant tout recours à la co-induction.

Ce travail a cependant fait apparaître quelques points faibles de cette approche à base de sémantique relationnelle. Par exemple, il n'est pas facile d'ajouter au calcul avec références des règles décrivant la récupération des cellules mémoire inutilisées (*garbage collection*). Aussi, prouver dans ce formalisme la sûreté forte d'un calcul polymorphe avec `callcc` est encore un problème ouvert. Certes, la propriété de sûreté faible suffit à se convaincre qu'un système de types est correct ; néanmoins, c'est sur la propriété de sûreté forte que reposent certaines techniques de compilation utilisant les informations de typage statique [50].

La principale difficulté technique de ce travail est le problème de non-conservativité auquel répond le système du chapitre 4. Le même problème se retrouve dans les systèmes d'effets avec typage des effets d'allocation, et je soupçonne que ce problème réapparaît à chaque fois qu'on annote les types fonctionnels par des informations sur les types de certains objets manipulés à l'intérieur de la fonction. La solution de ce problème a une justification sémantique assez simple, qui repose essentiellement sur la conjecture suivante : si une expression est bien typée en supposant deux de ses sous-expressions de types différents, alors ces deux sous-expressions ne peuvent pas communiquer à l'exécution ; en particulier, elles ne peuvent pas s'évaluer en la même référence. C'est évident si les deux sous-expressions sont de deux types de base différents, comme `int` et `bool` ; cela semble encore juste (quoique moins évident) si les deux sous-expressions ont pour types deux variables de types α et β différentes.

Ce qui est étonnant, c'est que ce "théorème de non-communication" et l'opération de simplification des types de fermetures qu'il justifie n'apparaissent nulle part dans le système du chapitre 4. Le système du chapitre 4 repose tout entier sur un raffinement de l'opération de généralisation des types, dont la justification est purement syntaxique. Les étiquettes placées sur les types fonctionnels peuvent être vues comme des constructeurs de types d'arité variable, et dont l'ordre et la multiplicité des types arguments n'a pas d'importance. Contrairement à ce qui se passe avec les constructeurs de types habituels, d'arité fixe, il est donc correct de généraliser des variables de types en-dessous d'une étiquette non généralisable. Chaque instanciation du type générique obtenu va simplement ajouter l'instance produite à l'étiquette non générique. La preuve de sûreté consiste pour l'essentiel à vérifier que cette opération garde correctement trace de tous les types avec lesquels les valeurs de la partie environnement d'une fermeture sont considérées. La preuve est un peu longue parce que le système du chapitre 4 est très bureaucratique ; mais c'est tout ce qu'elle établit : je ne vois pas de résultat de non-communication, même entre les lignes.

Il serait sans doute instructif de reformuler le système conservatif du chapitre 4 d'une manière qui "colle" mieux à l'intuition sémantique initiale. J'ai fait un certain nombre de tentatives dans cette direction, mais elles ont toutes échoué sur le point suivant : les systèmes de types obtenus ne sont pas stables par substitution. En effet, si on identifie deux variables de types α et β, deux expressions de types respectifs α et β, et donc ne communiquant pas, se mettent à avoir le même type, et donc semblent pouvoir communiquer. En d'autres termes, le critère de non-communication est le plus précis quand on considère des typages principaux ; il se perd dans le vague au fur et à mesure qu'on affaiblit le typage. C'est à ma connaissance un problème ouvert de tenir compte de la principalité d'un typage dans la preuve d'une propriété sémantique.

Bibliographie

[1] Hassan Aït-Kaci et Roger Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2(1):51–89, 1989.

[2] Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[3] Andrew W. Appel et David B. MacQueen. *Standard ML reference manual (preliminary)*. AT&T Bell Laboratories, 1989. Distribué avec Standard ML of New Jersey.

[4] Lennart Augustsson. A compiler for lazy ML. In *Lisp and Functional Programming 1984*, pp. 218–227. ACM Press, 1984.

[5] Henry G. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *Lisp and Functional Programming 1990*. ACM Press, 1990.

[6] Jean-Pierre Banâtre et Daniel Le Métayer. A new computational model and its discipline of programming. Rapport de recherche 566, INRIA, 1986.

[7] Dave Berry. The Edinburgh SML library. Rapport technique ECS-LFCS-91-148, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[8] Dave Berry, Robin Milner, et David N. Turner. A semantics for ML concurrency primitives. In *19th symposium Principles of Programming Languages*, pp. 119–129. ACM Press, 1992.

[9] Gérard Berry et Gérard Boudol. The chemical abstract machine. In *17th symposium Principles of Programming Languages*. ACM Press, 1990.

[10] Bernard Berthomieu et Thierry Le Sergent. Programming with behaviors in an ML framework: the syntax and semantics of LCS. In D. Sannella (rédacteur), *Programming languages and systems – ESOP '94*, Lecture Notes in Computer Science, volume 788, pp. 89–104. Springer, 1994.

[11] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[12] Luca Cardelli. Typeful programming. In E. J. Neuhold et M. Paul (rédacteurs), *Formal description of programming concepts*, pp. 431–507. Springer, 1989.

[13] Luca Cardelli et John C. Mitchell. Operations on records. In *Mathematical Foundations of Programming Semantics*, Lecture Notes in Computer Science, volume 442, pp. 22–52, 1989.

[14] Luca Cardelli et Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing surveys*, 17(4):471–522, 1985.

[15] Nicholas Carriero et David Gelerntner. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[16] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoet, et Gilles Kahn. Natural semantics on the computer. Rapport de recherche 416, INRIA, 1985.

[17] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, et Gilles Kahn. A simple applicative language: Mini-ML. Rapport technique 529, INRIA, 1986.

[18] Guy Cousineau, Pierre-Louis Curien, et Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[19] Guy Cousineau et Gérard Huet. The CAML primer. Rapport technique 122, INRIA, 1990.

[20] Patrick Cousot et Radhia Cousot. Inductive definitions, semantics and abstract interpretation. In *19th symposium Principles of Programming Languages*, pp. 83–94. ACM Press, 1992.

[21] Luis Damas. *Type assignment in programming languages.* Thèse de PhD, University of Edinburgh, 1985.

[22] Luis Damas et Robin Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pp. 207–212. ACM Press, 1982.

[23] Daniel de Rauglaudre. An implementation of monomorphic dynamics in ML using closures and references. Note, INRIA, projet Formel, 1991.

[24] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, et Benjamin Werner. The Coq proof assistant user's guide: version 5.6. Rapport technique 134, INRIA, 1991.

[25] Bruce F. Duba, Robert Harper, et David B. MacQueen. Typing first-class continuations in ML. In *18th symposium Principles of Programming Languages*, pp. 163–173. ACM Press, 1991.

[26] Matthias Felleisen et Daniel P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3):243–287, 1989.

[27] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, et Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[28] You-Chin Fuh et Prateek Mishra. Type inference with subtypes. In *ESOP '88*, Lecture Notes in Computer Science, volume 300, pp. 94–114. Springer Verlag, 1988.

[29] David K. Gifford et John M. Lucassen. Integrating functional and imperative programming. In *13th symposium Principles of Programming Languages*, pp. 28–38. ACM Press, 1986.

[30] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'État, Université Paris VII, 1972.

[31] Michael J. Gordon, Arthur J. Milner, et Christopher P. Wadsworth. *Edinburgh LCF*, Lecture Notes in Computer Science, volume 78. Springer, 1979.

[32] Carl A. Gunter et Didier Rémy. Ravl: A language for programming with records and variants. Soumis à publication, 1992.

[33] Carl A. Gunter et Dana S. Scott. Semantic domains. In van Leeuwen [95], pp. 635–674.

[34] Robert Harper. Introduction to Standard ML. Rapport technique ECS-LFCS-86-14, University of Edinburgh, 1986.

[35] Robert Harper et Mark Lillibridge. ML with `callcc` is unsound. Message de la liste de distribution `sml`, juillet 1991.

[36] Christopher T. Haynes, Daniel P. Friedman, et Mitchell Wand. Obtaining coroutines from continuations. *Computer Languages*, 11(3/4):143–153, 1986.

[37] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, 1985.

[38] Paul Hudak, Simon Peyton Jones, et Philip Wadler. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), 1992.

[39] Lalita A. Jategaonkar et John C. Mitchell. ML with extended pattern matching and subtypes. In *Lisp and Functional Programming 1988*, pp. 198–211. ACM Press, 1988.

[40] Pierre Jouvelot et David K. Gifford. Algebraic reconstruction of types and effects. In *18th symposium Principles of Programming Languages*, pp. 303–310. ACM Press, 1991.

[41] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information processing 74*, pp. 471–475. North-Holland, 1974.

[42] Gilles Kahn. Natural semantics. In K. Fuchi et M. Nivat (rédacteurs), *Programming of Future Generation Computers*, pp. 237–257. Elsevier, 1988.

[43] Gilles Kahn et David B. MacQueen. Coroutines and networks of parallel processes. Rapport de recherche 202, INRIA, 1976.

[44] Paris Kanellakis et John C. Mitchell. Polymorphic unification and ML typing. In *16th symposium Principles of Programming Languages*. ACM Press, 1989.

[45] Yves Lafont. Interaction nets. In *17th symposium Principles of Programming Languages*. ACM Press, 1990.

[46] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1964.

[47] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–165, 1966.

[48] Xavier Leroy. Une extension de Caml vers la programmation logique. Mémoire de première année, Ecole Normale Supérieure, 1988.

[49] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Rapport technique 117, INRIA, 1990.

[50] Xavier Leroy. Unboxed objects and polymorphic typing. In *19th symposium Principles of Programming Languages*, pp. 177–188. ACM Press, 1992.

[51] Xavier Leroy et Pierre Weis. Polymorphic type inference and assignment. In *18th symposium Principles of Programming Languages*, pp. 291–302. ACM Press, 1991.

[52] Barbara Liskov, Russell Atkinson, et Toby Bloom. *CLU reference manual*, Lecture Notes in Computer Science, volume 114. Springer, 1981.

[53] John M. Lucassen et David K. Gifford. Polymorphic effect systems. In *15th symposium Principles of Programming Languages*, pp. 47–57. ACM Press, 1988.

[54] David B. MacQueen. Modules for Standard ML. In Robert Harper, David B. MacQueen, et Robin Milner (rédacteurs), *Standard ML*. University of Edinburgh, technical report ECS LFCS 86-2, 1986.

[55] David B. MacQueen, Gordon Plotkin, et Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[56] David C. J. Matthews. Poly manual. Rapport technique 63, University of Cambridge, 1985.

[57] Michel Mauny. *Functional programming using Caml Light*. INRIA, 1991. Distribué avec Caml Light.

[58] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[59] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1990.

[60] Robin Milner. Message de la liste de distribution `sml`, juillet 1991.

[61] Robin Milner, Joachim Parrow, et David Walker. A calculus of mobile processes: part 1. Rapport de recherche ECS-LFCS-89-85, University of Edinburgh, 1989.

[62] Robin Milner et Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[63] Robin Milner et Mads Tofte. *Commentary on Standard ML*. The MIT Press, 1991.

[64] Robin Milner, Mads Tofte, Robert Harper, et David MacQueen. *The definition of Standard ML (revised)*. The MIT Press, 1997.

[65] John C. Mitchell. Coercion and type inference. In *11th symposium Principles of Programming Languages*, pp. 175–185. ACM Press, 1984.

[66] John C. Mitchell. Type systems for programming languages. In van Leeuwen [95], pp. 367–458.

[67] Vladimir Nabokov. *Ada or ardor, a family chronicle*. McGraw-Hill, 1969.

[68] Flemming Nielson. The typed λ-calculus with first-class processes. In *PARLE '89*, Lecture Notes in Computer Science, volume 366, pp. 357–373, 1989.

[69] Atsushi Ohori et Peter Buneman. Type inference in a database language. In *Lisp and Functional Programming 1988*, pp. 174–183. ACM Press, 1988.

[70] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[71] Lawrence C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[72] Simon L. Peyton-Jones. *The implementation of functional programming languages*. Prentice-Hall, 1987.

[73] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[74] Vincent Poirriez. *Intégration de fonctionnalités logiques dans un langage fonctionnel fortement typé: MLOG, une extension de ML*. Thèse de doctorat, Université Paris VII, 1991.

[75] Jonathan A. Rees et William Clinger. Revised^3 report on the algorithmic language Scheme. *ACM Sigplan Notices*, 21(12), décembre 1986.

[76] Didier Rémy. Records and variants as a natural extension of ML. In *16th symposium Principles of Programming Languages*, pp. 77–88. ACM Press, 1989.

[77] Didier Rémy. *Algèbres touffues. Application au typage polymorphe des objets enregistrements dans les langages fonctionnels*. Thèse de doctorat, Université Paris VII, 1991.

[78] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter et John C. Mitchell (rédacteurs), *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1993.

[79] John H. Reppy. First-class synchronous operations in Standard ML. Rapport technique TR 89-1068, Cornell University, 1989.

[80] John H. Reppy. CML: a higher-order concurrent language. *SIGPLAN Notices*, 6(26):293–305, 1991.

[81] John C. Reynolds. Toward a theory of type structure. In *Programming Symposium, Paris, 1974*, Lecture Notes in Computer Science, volume 19, pp. 408–425. Springer, 1974.

[82] John C. Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development (TAPSOFT 85)*, Lecture Notes in Computer Science, volume 185, pp. 97–138. Springer, 1985.

[83] John C. Reynolds. Preliminary design of the programming language Forsythe. Rapport technique CMU-CS-88-159, Carnegie Mellon University, 1988.

[84] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[85] François Rouaix. *ALCOOL-90: Typage de la surcharge dans un langage fonctionnel*. Thèse de doctorat, Université Paris 7, 1990.

[86] François Rouaix. Safe run-time overloading. In *17th symposium Principles of Programming Languages*. ACM Press, 1990.

[87] Robert Sedgewick. *Algorithms*. Addison-Wesley, deuxième édition, 1988.

[88] Gert Smolka. FRESH: a higher-order language with unification and multiple results. In Doug DeGroot et Gary Lindstrom (rédacteurs), *Logic Programming: Functions, Relations, and Equations*, pp. 469–524. Prentice-Hall, 1986.

[89] Jean-Pierre Talpin et Pierre Jouvelot. The type and effect discipline. In *Logic in Computer Science 1992*. IEEE Computer Society Press, 1992.

[90] Bent Thomsen. A calculus of higher-order communicationg systems. In *16th symposium Principles of Programming Languages*. ACM Press, 1989.

[91] Mads Tofte. Operational semantics and polymorphic type inference. Thèse de PhD CST-52-88, University of Edinburgh, 1988.

[92] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), 1990.

[93] David A. Turner. Miranda, a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture 1985*, Lecture Notes in Computer Science, volume 201, pp. 1–16. Springer, 1985.

[94] U.S. Department of Defense. *Reference manual for the ADA programming language*, 1983. ANSI-MIL-STD 1815A.

[95] Jan van Leeuwen (rédacteur). *Handbook of Theoretical Computer Science, volume B*. The MIT Press/Elsevier, 1990.

[96] Mitchell Wand. Continuation-based multiprocessing. In *Lisp and Functional Programming 1980*, pp. 19–28. ACM Press, 1980.

[97] Mitchell Wand. Complete type inference for simple objects. In *Logic in Computer Science 1987*, pp. 37–44. IEEE Computer Society Press, 1987.

[98] Pierre Weis et al. The CAML reference manual, version 2.6.1. Rapport technique 121, INRIA, 1990.

[99] Andrew K. Wright. Typing references by effect inference. In *European Symposium on Programming*, Lecture Notes in Computer Science, volume 582. Springer, 1992.

[100] Andrew K. Wright et Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Table des matières

Index